Ph.D. Dissertation

Design Techniques for Energy-Efficient Cache using STT-RAM

STT-RAM을 이용한 에너지 효율적인 캐시 설계 기술

February 2019

Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

Namhyung Kim

Ph.D. Dissertation

Design Techniques for Energy-Efficient Cache using STT-RAM

STT-RAM을 이용한 에너지 효율적인 캐시 설계 기술

February 2019

Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

Namhyung Kim

Design Techniques for Energy-Efficient Cache using STT-RAM

STT-RAM을 이용한 에너지 효율적인 캐시 설계 기술

지도교수 최 기 영

이 논문을 공학박사 학위논문으로 제출함

2018 년 11 월

서울대학교 대학원

전기 컴퓨터 공학부

김 남 형

김 남 형의 공학박사 학위논문을 인준함

2018 년 12 월

위 원 장	유승주	(인)
부위원장	최기영	(인)
위 원	김장우	(인)
위 원	이재욱	(인)
위 원	김경훈	(인)

Abstract

Over the last decade, the capacity of on-chip cache is continuously increased to mitigate the memory wall problem. However, SRAM, which is a dominant memory technology for caches, is not suitable for such a large cache because of its low density and large static power. One way to mitigate these downsides of the SRAM cache is replacing SRAM with a more efficient memory technology. Spin-Transfer Torque RAM (STT-RAM), one of the emerging memory technology, is a promising candidate for the alternative of SRAM. As a substitute of SRAM, STT-RAM can compensate drawbacks of SRAM with its non-volatility and small cell size. However, STT-RAM has poor write characteristics such as high write energy and long write latency and thus simply replacing SRAM to STT-RAM increases cache energy. To overcome those poor write characteristics of STT-RAM, this dissertation explores three different design techniques for energy-efficient cache using STT-RAM.

The first part of the dissertation focuses on combining STT-RAM with exclusive cache hierarchy. Exclusive caches are known to provide higher effective cache capacity than inclusive caches by removing duplicated copies of cache blocks across hierarchies. However, in exclusive cache hierarchies, every block evicted from the upper-level cache is written back to the last-level cache regardless of its dirtiness thereby incurring extra write overhead. This makes it challenging to use STT-RAM for exclusive last-level caches due to its high write energy and long write latency. To mitigate this problem, we design an SRAM/STT-RAM hybrid cache architecture based on reuse distance prediction.

The second part of the dissertation explores trade-offs in the design of volatile STT-RAM cache. Due to the inefficient write operation of STT-RAM, various solutions have

been proposed to tackle this inefficiency. One of the proposed solutions is redesigning STT-RAM cell for better write characteristics at the cost of shortened retention time (i.e., volatile STT-RAM). Since the retention failure of STT-RAM has a stochastic property, an extra overhead of periodic scrubbing with error correcting code (ECC) is required to tolerate the failure. With an analysis based on analytic STT-RAM model, we have conducted extensive experiments on various volatile STT-RAM cache design parameters including scrubbing period, ECC strength, and target failure rate. The experimental results show the impact of the parameter variations on last-level cache energy and performance and provide a guideline for designing a volatile STT-RAM with ECC and scrubbing.

The last part of the dissertation proposes Benzene, an energy-efficient distributed SRAM/STT-RAM hybrid cache architecture for manycore systems running multiple applications. It is based on the observation that a naïve application of hybrid cache techniques to distributed caches in a manycore architecture suffers from limited energy reduction due to uneven utilization of scarce SRAM. We propose two-level optimization techniques: intra-bank and inter-bank. Intra-bank optimization leverages highly-associative cache design, achieving more uniform distribution of writes within a bank. Inter-bank optimization evenly balances the amount of write-intensive data across the banks.

Keywords: Computer Architecture, Memory System, Cache, Energy-Efficient, Non-Volatile Memory, STT-RAM

Student Number: 2013-20751

Contents

Abstract **i**

Contents **iii**

List of Figures **vii**

List of Tables **xi**

Chapter 1 Introduction **1**

1.1 Exclusive Last-Level Hybrid Cache 2

1.2 Designing Volatile STT-RAM Cache 4

1.3 Distributed Hybrid Cache 5

Chapter 2 Background **9**

2.1 STT-RAM 9

2.1.1 Thermal Stability 10

2.1.2 Read and Write Operation of STT-RAM 11

2.1.3 Failures of STT-RAM 11

2.1.4 Volatile STT-RAM 13

2.1.5 Related Work 14

2.2 Exclusive Last-Level Hybrid Cache 18
2.2.1 Cache Hierarchies 18
2.2.2 Related Work 19
2.3 Distributed Hybrid Cache 21
2.3.1 Prediction Hybrid Cache. 21
2.3.2 Distributed Cache Partitioning 22
2.3.3 Related Work 23

Chapter 3 Exclusive Last-Level Hybrid Cache 27

3.1 Motivation 27
3.1.1 Exclusive Cache Hierarchy 27
3.1.2 Reuse Distance 29
3.2 Architecture 30
3.2.1 Reuse Distance Predictor 30
3.2.2 Hybrid Cache Architecture 32
3.3 Evaluation 34
3.3.1 Methodology 34
3.3.2 LLC Energy Consumption 35
3.3.3 Main Memory Energy Consumption 38
3.3.4 Performance 39
3.3.5 Area Overhead 39
3.4 Summary 39

Chapter 4 Designing Volatile STT-RAM Cache 41

4.1 Analysis 41
4.1.1 Retention Failure of a Volatile STT-RAM Cell 41
4.1.2 Memory Array Design 43
4.2 Evaluation 45

4.2.1 Methodology 45
4.2.2 Last-Level Cache Energy 46
4.2.3 Performance 51
4.3 Summary 52

Chapter 5 Distributed Hybrid Cache **55**
5.1 Motivation 55
5.2 Architecture 58
5.2.1 Intra-Bank Optimization 59
5.2.2 Inter-Bank Optimization 63
5.2.3 Other Optimizations 67
5.3 Evaluation Methodology 69
5.4 Evaluation Results 73
5.4.1 Energy Consumption and Performance 73
5.4.2 Analysis of Intra-bank Optimization 76
5.4.3 Analysis of Inter-bank Optimization 78
5.4.4 Impact of Inter-Bank Optimization on Network Energy 79
5.4.5 Sensitivity Analysis 80
5.4.6 Implementation Overhead 81
5.5 Summary 82

Chapter 6 Conculsion **85**

Bibliography **88**

초록 **101**

List of Figures

Figure 2.1 MTJ and STT-RAM cell structure. 9

Figure 3.1 The number of LLC writes in the exclusive cache hierarchy normalized to that of the inclusive cache hierarchy. 28

Figure 3.2 Reuse distance distribution. 29

Figure 3.3 Overview of the proposed architecture. 30

Figure 3.4 Energy consumption and performance of our architecture normalized to the STT-RAM baseline. 36

Figure 3.5 Energy breakdown of STT-RAM-Baseline (left) and Hybrid-Bypass (right). 38

Figure 4.1 Retention failure characteristics of an STT-RAM cell. 42

Figure 4.2 Relationship between memory array design parameters. . . . 43

Figure 4.3 Relationship between target failure rate and thermal stability. 44

Figure 4.4 Last-level cache energy consumption and performance of various configurations. 47

Figure 4.5 Last-level cache energy consumption on different target failure rates. 49

Figure 4.6 Last-level cache energy consumption for SECDED_100ms normalized to non-volatile STT-RAM baseline. 50
Figure 4.7 Performance for different write latency of STT-RAM. 52

Figure 5.1 The distribution of SRAM/STT-RAM writes across different sets in a set-associative cache. 56
Figure 5.2 The distribution of writes across different banks in a distributed cache architecture (i.e., Jigsaw). 57
Figure 5.3 Overall architecture. 58
Figure 5.4 Intra-bank optimization. W_1 to W_3 are write-intensive blocks, while R_1 indicates a read-intensive block. 61
Figure 5.5 Decision tree for threshold adjustment in Benzene. Decision tree in PHC is shown in the dotted box 68
Figure 5.6 LLC energy consumption (above) and performance (below) of our architecture compared to the SRAM and STT-RAM baseline. All results are normalized to the STT-RAM baseline. 73
Figure 5.7 LLC energy breakdown. 'Relocation' represents the energy overhead of additional read/write operations caused by the cuckoo hashing mechanism of ZCache. 74
Figure 5.8 LLC dynamic energy (left) and miss rate (right) of Benzene-Intra and Benzene (normalized to STT-RAM baseline). . . . 75
Figure 5.9 Histogram of SRAM write distribution across different sets in a set-associative cache (above) and a highly-associative cache with a tag-to-data pointer table (below). 77
Figure 5.10 Histogram of write distribution across different banks in a distributed cache before (above) and after (below) applying our inter-bank optimization. 78

Figure 5.11 Normalized energy consumption of LLC and on-chip network in Benzene-Intra (left) and Benzene (right). 79
Figure 5.12 Impact of the hop count constraint on LLC energy (left), network energy (middle), and performance (right). The results are normalized to Benzene-Intra (Intra in the figure). 80
Figure 5.13 The impact of adaptive inter-bank optimization on LLC energy consumption and performance. Benzene without adaptive inter-bank optimization (left) and Benzene (right), compared to Benzene-Intra. 81
Figure 5.14 The impact of partitioning-aware sampling and better threshold adjustment on LLC miss rate (left) and speedup (right). The figure shows comparison among the STT-RAM baseline (**S**), Benzene-Intra without partitioning-aware sampling or threshold adjustment (**B**), Benzene-Intra with partitioning-aware sampling (**P**), Benzene-Intra with threshold adjustment (**T**), and Benzene-Intra with partitioning-aware sampling and threshold adjustment (**P+T**). 82

List of Tables

Table 3.1 Characteristics of the SRAM, STT-RAM, and Hybrid Cache . 35

Table 4.1 Characteristics of ECC encoder/decoder 46

Table 5.1 Configuration of the Simulated System 70

Table 5.2 Characteristics of an LLC Tile 70

Table 5.3 Applications from SPEC CPU2006 71

Table 5.4 Workload characteristics 72

Chapter 1

Introduction

In modern processors, the performance of cores has continuously increased whereas the main memory performance has been improved very slowly compared to the cores. Thus, main memory access becomes the main bottleneck of the system performance [1]. Therefore, the capacity of on-chip cache has continuously increased to filter out more maim memory accesses, which leads to better performance and energy efficiency. In addition, the trend of increasing number of cores calls for more efficient and larger caches to reduce costly main memory accesses. For instance, Intel SkyLake-SP Xeon server processor [2] has a 38.5MB last-level cache (LLC) which occupies a significant portion of the chip area.

However, conventional charge-based memory technologies (e.g., SRAM and DRAM) is not suitable for LLCs since those memory technologies need power supply to keep data alive. Contrary to them, Spin-Transfer Torque RAM (STT-RAM), an emerging memory technology, stores its information in the form of magnetization direction, and thus, the data stored in STT-RAM persist even without power supply (i.e., non-volatility). Due to this, STT-RAM consumes very low static energy compared to conventional

charge-based technologies.

Despite this advantages of STT-RAM over SRAM, the drawback of STT-RAM caches is that they consume much higher write energy with longer write latency than its SRAM counterpart. To alleviate the impact of write inefficiency of STT-RAM, this dissertation proposes techniques for designing energy-efficient caches with STT-RAM in three parts: hybrid cache architectures for exclusive cache hierarchy, utilizing volatile STT-RAM, and distributed hybrid cache design for manycore systems.

1.1 Exclusive Last-Level Hybrid Cache

As explained before, the capacity of on-chip cache has continuously increased in order to filter out more main memory accesses, which leads to better performance and energy efficiency. However, conventional SRAM-based inclusive cache hierarchies may not be the optimal design in that (1) increasing SRAM size is detrimental in terms of area and energy efficiency and (2) a significant portion of on-chip cache capacity is wasted by storing multiple copies of a single cache block across multiple levels of caches.

Regarding the first inefficiency, the recent advancement of emerging memory technologies is opening up the possibility of enlarging LLCs in a more efficient manner. In particular, Spin-Transfer Torque RAM (STT-RAM) is gaining attention as a promising alternative to SRAM for LLCs. Compared to SRAM, STT-RAM shows much lower static power and smaller cell size due to its unique cell structures based on non-volatile, magnetic-based information storage. With these benefits, STT-RAM caches have been projected to realize very large on-chip LLCs while minimizing its impact on area and energy [3, 4].

Another dimension of increasing the on-chip cache efficiency is to employ better cache hierarchy management. In inclusive cache hierarchies, which are the most widely used, the inclusion property mandates each upper-level cache block (e.g., L1 cache

block) to be duplicated across all lower-level caches (e.g., last-level cache), thereby degrading on-chip storage efficiency. On the contrary, exclusive caches are free from such overhead as they store each block in only a single place. The performance gap between them is expected to be wider in modern CMPs due to deeper cache levels and larger upper-level caches [5]. Because of these advantages, several commercial products already adopted exclusive cache hierarchies, including AMD's desktop and server processors [6, 7].

As both STT-RAM and exclusive caches provide advantages over conventional SRAM-based inclusive caches, this work for the first time explores STT-RAM-based exclusive LLC design. Our finding is that, while both STT-RAM and exclusive caches can improve the efficiency of LLCs, simply replacing SRAM with STT-RAM is very inefficient in exclusive LLCs. This is because enforcing the exclusion property greatly increases LLC writes (65% more writes compared to inclusive caches according to our experiments), which is harmful to STT-RAM caches whose drawback is high write energy.

To mitigate this, we propose a hybrid cache architecture composed of SRAM and STT-RAM based on reuse distance prediction [8]. The key idea is to identify cache blocks with near/far reuse and then utilize the information to determine the block placement or bypassing. This greatly improves energy efficiency of STT-RAM exclusive LLCs, which has not been possible with existing approaches for STT-RAM inclusive/non-inclusive caches due to the fundamental differences in cache behavior on hits/misses between inclusive/non-inclusive and exclusive caches.

The details of this first part of the work are described in Section 3 of this dissertation. Following is the summary of the contributions:

- For the first time, we evaluate STT-RAM in the context of exclusive cache hierarchies and identify the key challenges in designing an energy-efficient

exclusive LLC based on STT-RAM.

- We propose a hybrid cache architecture for exclusive STT-RAM LLCs based on PC-directed reuse distance prediction. This is the first approach that utilizes the reuse distance prediction to improve energy efficiency of caches without any software modification.
- We evaluate our architecture based on a cycle-level simulator and show that it reduces energy consumption of the LLC by 55% compared to the STT-RAM baseline with slight performance improvement.

1.2 Designing Volatile STT-RAM Cache

To overcome the aforementioned poor write characteristics of STT-RAM, relaxing non-volatility of STT-RAM (thus making STT-RAM volatile) has been proposed [9]. By reducing STT-RAM cell area, thermal stability (Δ) which represents height of thermal barrier for bit flipping is decreased to make STT-RAM volatile. As thermal stability decreases, write energy and latency is also decreased. However, lowering thermal stability makes bit flipping by thermal noise (retention failure) easier and thus decreases retention time. Shortening retention time increases possibility of losing data, and therefore, architectural support to reliably store data in volatile STT-RAM is necessary to utilize the better write characteristics of volatile STT-RAM.

In the second part (Section 4), based on the analytic model [10, 11] of volatile STT-RAM, we analyze and evaluate relationship among thermal stability, error correcting capability, scrubbing overhead, and target failure rate. Then we provide a guideline of designing volatile STT-RAM cell and memory array in terms of energy efficiency and performance [12]. Although there are some previous researches on utilizing volatile STT-RAM, they only utilize it with a fixed configuration to improve performance [13, 14], write energy [15, 16], and density [17]. Unlike those researches, this work explores for

the first time how different volatile STT-RAM design parameters affect cache energy and system performance.

1.3 Distributed Hybrid Cache

Over the last decade, processor speed improvement has been achieved by increasing the number of cores rather than frequency scaling due to the limited amount of instruction-level parallelism and power inefficiency in improving performance with scaling the clock speed. Accordingly, processor architectures with tens of or more cores, called manycore architecture, are proposed and have become popular in both academia and industry (e.g., Tilera TILE64 [18] and Intel Xeon Phi [19]). As these architectures are designed to run large numbers of applications simultaneously to fully utilize all cores in the system, they often incorporate tens of megabytes of LLCs to efficiently handle frequent memory accesses from the applications.

However, there are two critical issues in using large shared on-chip LLC with manycore systems—energy efficiency and quality of service (QoS) guarantee. First, SRAM, which is the most popular memory technology for constructing on-chip caches, is not suitable for large LLC due to its high static energy and low density. This can be alleviated by employing a new memory technology, such as STT-RAM, that provides much lower static energy and higher density thereby being much more efficient in constructing large on-chip caches. Despite these advantages over SRAM, STT-RAM suffers from poor write characteristics in terms of energy and latency. To mitigate the problem, recent work proposed hybrid caches [20, 21], which combines small SRAM with large STT-RAM to service write-intensive data with the SRAM. Second, as LLC in manycore systems is shared across more and more applications running at the same time, interference among applications (e.g., thrashing) significantly degrades cache efficiency, resulting in failure to guarantee QoS. To mitigate such an interference

problem in manycore systems, distributed cache partitioning [22, 23] was proposed to isolate the impact of each application on the LLC.

While hybrid cache architectures and cache partitioning techniques for manycore systems are widely explored, none of the prior work considers distributed hybrid caches with manycore cache partitioning. More importantly, we observe that simply combining existing techniques together is not efficient enough because the two problems interact with each other. For example, the block allocation policy in hybrid caches determines cache blocks to be allocated to SRAM instead of STT-RAM, but its effectiveness highly depends also on how applications are mapped to each cache bank (determined by distributed cache partitioning).

In this work, we propose *Benzene*, a distributed hybrid cache architecture for manycore systems [24]. Each distributed LLC slice is composed of an SRAM/STT-RAM hybrid cache and stores write-intensive data in SRAM to reduce costly STT-RAM writes. Under this organization, however, we observe significant write imbalance within a bank and across the banks. This write imbalance harms SRAM utilization, which is the key factor of energy efficiency in hybrid caches. To alleviate the problem, Benzene performs two key optimizations: (1) intra-bank optimization, which evenly distributes writes across different sets within each bank, and (2) inter-bank optimization, which balances the number of write-intensive data in each bank by leveraging cache block placement in distributed caches. With these two optimizations, Benzene utilizes SRAM more efficiently and reduces cache energy by 47.1% on average with minimal performance overhead.

The details of this last part of the work are described in Section 5 of this dissertation. Following is the summary of the contributions.

- For the first time, we explore distributed hybrid cache design for manycore systems with cache partitioning and observe the write imbalance problem in two

different levels, which incurs inefficiency in hybrid cache management.

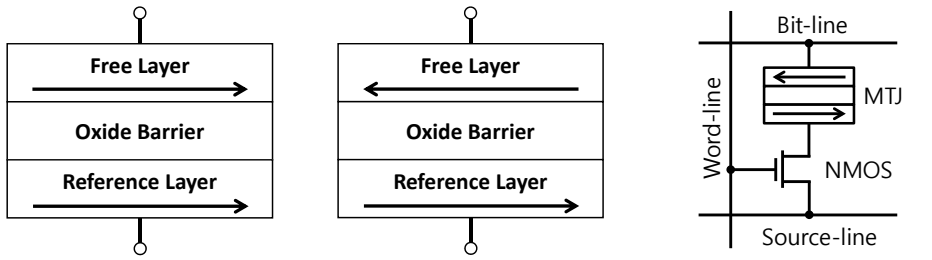
- To address the write imbalance problem, we propose Benzene, a distributed hybrid cache architecture for manycore systems. It is composed of two levels of optimizations for distributed hybrid caches that aim at evenly balancing the number of writes to each bank and each cache set.
- We evaluate Benzene based on an architectural simulator and show that it achieves a 47.1% LLC energy reduction compared to the state-of-the-art distributed cache architecture.

Chapter 2

Background

2.1 STT-RAM

Unlike conventional charge-based memory technologies (i.e., SRAM, DRAM) which store data in the capacitor, STT-RAM stores data in a special structure called Magnetic Tunnel Junction (MTJ). An MTJ consists of two ferromagnetic layers (reference layer and free layer) and one oxide layer between them. Each ferromagnetic layer has its own magnetic direction. The reference layer has a fixed magnetic direction and the free



(a) Parallel state (b) Anti-parallel state (c) STT-RAM cell structure

Figure 2.1 MTJ and STT-RAM cell structure.

layer has its own magnetic direction that can be flipped by the current flowing through the MTJ. According to the magnetic direction of free layer, there are two different states (parallel state and anti-parallel state) of MTJ as shown in Figure 2.1 and resistance of an MTJ differs as state of the MTJ changes—low resistance in parallel state and high resistance in anti-parallel state. The two different resistance values represent the stored data in the MTJ.

Based on such MTJ, one-transistor-one-MTJ (1T-1MTJ) is the most popular structure of an STT-RAM cell. As can be seen in Figure 2.1c, it consists of one NMOS transistor and one MTJ. Word-line is used for bit selection and then appropriate voltage is applied to the bit-line and source-line to write new data or read stored data.

Static power of STT-RAM is much lower than SRAM since it is unnecessary to consume extra power for maintaining the stored data because of its non-volatility. On the contrary, write characteristics such as write energy and write latency are the main drawbacks of STT-RAM.

2.1.1 Thermal Stability

Thermal stability (Δ) is one of STT-RAM cell design parameters. It is proportional to energy barrier between the two MTJ states (parallel and anti-parallel states). Thermal stability is defined as [25]:

$$\Delta = \frac{E_b}{k_B T} = \frac{H_K M_S V}{2 k_B T} \tag{2.1}$$

Where, E_b is energy barrier, k_B is the Boltzmann constant and T is temperature. H_K is anisotropy field, M_S is saturation magnetization and V is volume of the MTJ.

As can be seen in (2.1), thermal stability is proportional to energy barrier and the energy barrier is also proportional to the volume of the MTJ. Therefore, thermal stability can be controlled by how the cell is designed. Energy barrier is minimum energy

required to change state of MTJ hence it affects write characteristics. Thus, STT-RAM with better write characteristics can be obtained by reducing thermal stability.

2.1.2 Read and Write Operation of STT-RAM

Writing a data to an STT-RAM cell is performed by a state transition of an MTJ into parallel state (P) or anti-parallel state (AP). To change the MTJ state from anti-parallel state to parallel state (AP→P), a specific amount of current (I_{WR}) flows from the bit-line to the source-line and flips the magnetic direction of the free layer to be the same as that of the reference layer. On the contrary, current flowing from the source-line to the bit-line changes MTJ state from parallel state to anti-parallel state (P→AP). Two different state transitions use the same current path but in different directions. As mentioned before, write operation (switching MTJ state) of STT-RAM is a stochastic process and its probability is modeled as a function of thermal stability (Δ), switching pulse width (t_{sw}), applied current (I_C), and critical switching current (I_{C0}) [26]:

$$P_{sw} = 1 - exp\left(-\frac{t_{sw}}{\tau_0} e^{-\Delta\left(1-\frac{I_C}{I_{C0}}\right)}\right) \tag{2.2}$$

To read a data stored in an STT-RAM cell, identifying whether resistance of MTJ is high or low is required. It can be done by sensing the amount of flowing current (I_{RD}, lower than write current) through the MTJ when the specific voltage is applied between the bit-line and the source-line. If the sensed current is higher than a given reference, the STT-RAM cell is identified as a parallel state (low resistance); otherwise, it is identified as an anti-parallel state (high resistance). Read operation of STT-RAM uses same current path with write operation, but only in a single direction.

2.1.3 Failures of STT-RAM

Write Failure. Write failure occurs when intended switching does not happen properly after flowing write current. This cannot be prevented since write operation is a

stochastic process. As in (2.2), switching probability is always less than one even though sufficiently large current (I_C) flows for sufficiently long time (t_{sw}). To mitigate write failure of STT-RAM, recursive write-read-verify method [27] was proposed, which repeatedly apply a write pulse until successful write is monitored.

Read Failure. There are two different kinds of read failure in STT-RAM, read decision failure and read disturb failure. Although the same voltage is applied to bit-line and source-line of different STT-RAM cells, current through each STT-RAM cell may differ according to the variation of STT-RAM cell. Such a read current variation may cause a wrong decision during the read process resulting in read decision failure. Read disturb failure is an undesirable bit flip due to the read current. Since read and write operation of STT-RAM use the same current path, unintended bit flip may occur by read current. Even with low read current, probability of bit flipping is still non-zero although it is very low. (See (2.2))

Retention Failure. Retention failure is an unwanted bit flip due to thermal noise without switching current ($I_C = 0$). STT-RAM retention failure is a stochastic process which has behavior similar to SRAM soft error caused by cosmic rays. The probability of retention failure of a cell can be derived from (2.2) when $I_C = 0$ and is modeled as a function of thermal stability (Δ) and time (t) elapsed since the bit is written [10]:

$$P_{ret\_fail} = 1 - exp\left(\frac{-t}{exp(\Delta)}\right) \tag{2.3}$$

As shown in (2.3), error rate increases exponentially as Δ scales down (error probability decreases rapidly toward 0 as Δ scales up). Error probability also increases as time flows.

From (2.1) and (2.3), we can see that retention failure rate can be controlled by cell design. For example, size of the MTJ affects thermal stability (Δ) and it affects

retention failure rate. While error probability of a cell increases linearly with time, retention failure rate changes exponentially as thermal stability changes. In other words, small difference in thermal stability makes huge difference in retention failure rate.

2.1.4 Volatile STT-RAM

Volatile STT-RAM provides better write characteristics than conventional non-volatile STT-RAM at the cost of shorter retention time. Volatile STT-RAM can be designed with reduced cell area. From (2.1), reduced cell area lowers thermal stability and energy barrier and thus it also improves write characteristics. However, (2.3) indicates that reducing thermal stability makes retention failure rate higher. The improvement of write characteristics and the reduction of retention failure can be achieved by controlling thermal stability in opposite directions, and thus volatile STT-RAM should be designed while considering the trade-offs between them.

ECC and Scrubbing. To tolerate increased error probability for volatile STT-RAM, an architectural support is required. DRAM-style refresh has been proposed in the previous work [9, 13, 14]. However, retention time of volatile STT-RAM is neither deterministic nor predictable, and thus DRAM-style refresh is not suitable for mitigating STT-RAM retention failure [10, 11]. ECC and periodic scrubbing, which are used for SRAM soft errors [28, 29], can be employed for maintaining data reliably. Every scrubbing operation reads whole data in memory array one by one and checks to see if the data is correct. If error is detected then it is corrected and written back in the same place; otherwise write back operation is not necessary. Scrubbing period and ECC strength should be determined carefully. The scrubbing period should be short enough and the ECC should be strong enough so that errors are not accumulated. On the other hand, too frequent scrubbing may increase the system performance overhead, energy consumption, and even wearing of STT-RAM. Typically, the failure rate can be

set extremely low, and thus the correction and write back operation occurs very rarely. Surely, using stronger ECC and/or shorter scrubbing period makes volatile STT-RAM more tolerable while overhead of ECC and scrubbing is increased.

Memory Array. As explained in the previous section, ECC and periodic scrubbing are mandatory for the reliability of volatile STT-RAM. Designing a memory array with ECC and scrubbing is complex since ECC and scrubbing depends on each other; ECC allows errors per its granularity up to its correcting capacity and scrubbing should be repeated before there occur too many errors to be handled by the ECC. Therefore the STT-RAM cell (thermal stability) should be designed carefully considering memory array size, ECC strength, scrubbing period, and target reliability. The following equation shows how these parameters are related for proper operation [11].

$$\Delta \geq \frac{1}{1+k}\left[\sum_{i=0}^{k} \ln\left(\frac{m-i}{1+i}\right) + \ln\left(\frac{N}{\tau_0 F}\right) + k\ln\left(\frac{t_{ref}}{\tau_0}\right)\right] \tag{2.4}$$

Minimum thermal stability (Δ) can be calculated from (2.4) when memory array is designed as $N \times m$-bit array with k-bit error correction code, target failure rate F and scrubbed periodically with period t_{ref}.

2.1.5 Related Work

To mitigate the high write overhead of STT-RAM caches, various device-/circuit-/ architecture-level solutions are proposed. This section provides brief introduction to them.

Volatile STT-RAM. One of the most popular device-level solutions is volatile STT-RAM which is proposed by Smullen et al. [9]. They optimize STT-RAM cells for less write energy and latency by shortening retention time. Therefore, extra overhead is required to maintain data stored in volatile STT-RAM.

The cache architectures with volatile STT-RAM are also presented with its better write characteristics while minimizing extra overhead for its reliability. Sun et al. [14] proposed cache hierarchy consists of only STT-RAM with different retention time and utilize them in each cache level according to the performance requirement and data access pattern of each level. A modified DRAM-style refreshing scheme is proposed to reduce the overhead. It tracks lifetime of each data in cache and refreshes only the data that stay longer than the retention time. Jog et al. [13], based on their application characterization, provides optimal retention time for volatile STT-RAM caches. Instead of refresh all data in cache, it only refreshes few most recently used data and discards the others after retention time. Li et al. [30] utilizes prior knowledge from compiler to reduce the number of refresh operations.

These work are based on assumption that STT-RAM has fixed retention time and adopts DRAM-style read-and-write refresh to mitigate retention failure of volatile STT-RAM. However, as explained in Section 2.1.4, retention failure of STT-RAM can occur at any time due to its stochastic process. The DRAM-style refresh used in these work is not able to improve reliability of volatile STT-RAM. Instead, *periodic scrubbing with error correcting code (ECC)* is suggested [10, 11].

Redundant Write Elimination. The most of circuit-level solutions are on the basis of the observation that the content of the new data is same as the original data in high probability (i.e., value locality). Due to the value locality, it is possible to update only a part of the data and thus eliminate write operations.

Zhou et al. [31] presents *early write termination* that eliminates redundant bit-writes by terminating write operation in an early stage if the value already stored in MTJ is same as the new value. Bi et al. [32] and Zheng et al. [33] proposed *verify-one-while-write* and *variable-energy write*, respectively. Both of them use the property that writing STT-RAM is a stochastic process. Due to the stochastic property, switching time of

each bit flip operation is not fixed. Utilizing this characteristic, instead of using uniform current pulses, these two approaches cut off write current when successful write is monitored thus reduce write energy and write latency. These two works differ in the granularity of the monitoring. Variable-energy write allows bit-wise monitoring while verify-one-while-write can only monitor at the granularity of word.

Hybrid Caches. The key idea of hybrid caches [20, 21, 34–41] is to combine small SRAM and large STT-RAM and allocate write-intensive data to the SRAM. It allows us to take advantage of low static energy of STT-RAM and low write energy of SRAM at the same time. In this style of architecture, energy efficiency of caches is determined largely by how efficiently we can utilize the small SRAM for absorbing writes that would otherwise be made to STT-RAM. Therefore, better block placement policy is key to the energy efficient hybrid cache architecture.

To utilize scarce SRAM more efficiently, various allocation policies which make decision for data placement between SRAM and STT-RAM are proposed. The most naïve approaches [21, 34–36, 38–40] are to determine the block placement of each cache block according to the type of the instruction that generates the cache miss (e.g., SRAM on a store miss and STT-RAM on a load miss). This is based on the assumption that data loaded by store (or load) instructions are more likely to receive writes (or read) in the future. Chen et al. [37] allocates data to SRAM or STT-RAM based on both the hints to identify write-intensive data generated from compiler and the runtime cache behavior (e.g., cache pressure). Wang et al. [41] categorizes write accesses into three classes (i.e., writes by cores, by prefetch, by demand miss) and places data into SRAM or STT-RAM regarding the access pattern of each class.

Since such block placement may be incorrect, many of them combine their approach with *migration*, where blocks with frequent writes/reads are migrated to SRAM/STT-RAM to amend imperfect initial placement. Sun et al. [35] migrates data in STT-RAM

to SRAM when a cache block is accessed by two successive write operations. Wu et al. [34] proposed to swap data which receives two consecutive read/write hits in SRAM/ STT-RAM with LRU data in opposite region. Li et al. [40] adopts similar approach based on the number of reads (or writes) accesses to each cache block.

However, migration is known to incur a significant energy overhead because each migration operation needs one read and one write to move a block from one region to another. Since the researches mentioned before migrate data based on the type of accesses, read followed by write or write followed by read may generate excessive migration which offsets benefits from them. To reduce frequent migration, *migration-aware compilation* [36] and *compiler-assisted preferred cache* [39] are proposed. The former rearranges data layout at compile time to make data accessed by same type consecutively. The latter identifies migration-intensive data also at compile time and prioritize them to be allocated in SRAM.

The aforementioned block placement policies are suboptimal therefore they need migration with overhead and/or compiler assist which requires recompilation for each architecture. Ahn et al. [20] proposed prediction hybrid cache (PHC) that accurately predict write-intensive data and thus eliminates need for extra support. PHC predicts write intensity of each data at the time of data loading and decides where to place data with the prediction result. To predict write intensity, they define a cost model that measures write intensity of each data and correlate it with the instruction which causes miss.

Li et al. [38] proposed dual associative hybrid cache that allows neighbor cache sets share their SRAM blocks. Since hybrid caches have limited number of SRAM ways in a set to minimize static energy consumption and those few SRAM ways are only able to serve write-intensive data allocated to corresponding cache set, they are failed to utilize scarce SRAM resources efficiently when there are non-uniformity of write intensity among cache sets. Sharing SRAM ways improves chance to allocate write-intensive

data into SRAM, thereby achieves higher energy effiency.

STT-RAM Cache Bypass. The other way to reduce write operation of STT-RAM is to eliminate unnecessary writes. Wang et al. [42] proposed an obstruction-aware cache management, which improves performance by bypassing cache blocks from cores that generate a large number of writes and thus incur high bank contention. Although it reduces the LLC energy consumption, it greatly increases main memory accesses since it does not consider reuse information in bypass decision.

Ahn et al. [43] osbserved that significant amount of data written in last-level cache will not be re-referenced again. Therfore, those write operations are identified as unnecessary writes (i.e., dead write) and can be bypassed without increasing the number of misses. The bypass decision is made based on the prediction result from a dead write predictor presented in this work. The dead write predictor associates dead blocks with the instruction address that touches the cache block.

2.2 Exclusive Last-Level Hybrid Cache

2.2.1 Cache Hierarchies

There are some different types of cache hierarchy management policy which determines inclusion policy between different levels in cache hierarchy. Inclusive cache hierarchies guarantee that data stored in upper-level cache is also present in lower-level cache and thus cache hierarchy contains duplicated data across different cache level. Therfore, the total capacity of inclusive cache is same as the size of LLC.

To maintain the inclusion property, when data in lower-level cache is evicted, the corresponding data should be invalidated in upper-level cache (i.e., back-invalidation). Such an inclusion property makes coherence protocol simpler since miss from lower-level cache assures the data is not present in the upper-level cache. On the other hand,

non-inclusive caches do not guarantee inclusion property and thus there is no need for back-invalidation. Intel processors [44] adopt three-level cache hierarchy and L2 is non-inclusive to L1 cache. For LLC, they used to be inclusive to upper-level caches but LLC in the most recent server processors (i.e., Skylake-X) are non-inclusive to them.

Unlike inclusive cache hierarchies which store the same duplicated data in multiple levels of caches, exclusive cache hierarchies enforce only one location per cache block (exclusion property) to maximize cache efficiency. Therefore, the exclusive caches show better performance than inclusive caches, as the relative size of upper-level cache compared to lower-level cache becomes larger [5]. However, exclusive caches suffer from more on-chip traffic due to the clean victim from upper-level caches and coherence protocol becomes more complicated than inclusive caches with coherence directory or snooping upper-level caches. Contrary to the Intel processors, LLC in AMD processors adopts exclusive caches [6, 7].

2.2.2 Related Work

Cache Hierarchies. As explained in section 2.2.1, different cache hierarchies (inclusive/non-inclusive/exclusive cache) have different pros and cons. As a result, two major processor design companies, Intel and AMD, have made different design choices and many researchers explored different cache hierarchies and proposed cache architectures to take advantages of each cache hierarchies.

Zheng et al. [5] evaluated the impact on system performance of exclusive caches compared to that of inclusive caches with various cache size. The experimental result shows that exclusive caches show better performance in most applications with higher hit rate.

Sim et al. [45] identified that different applications prefer different inclusion policy (e.g., non-inclusive or exclusive) depending on their characteristics. The applications which are beneficial with larger cache capacity favors exclusive cache. On the contrary,

other applications prefer non-inclusive due to their fewer on-chip traffic. Based on the observation, *FLEXclusion* is proposed to dynamically adopt inclusion policy according to the requirement of each application.

Jaleel et al. [46] discovered that a shared LLC may be a bottleneck of the system performance for applications whose working set is larger than the size of private cache. They suggested to increase the size of private caches in the hierarchy rather than to increase the size of shared LLC. By increasing private upper-level cache size, relative size of shared LLC becomes smaller and thus significant cache capacity is wasted because of data duplication in inclusive cache. Therefore, in this work, they proposed to relax inclusion and build *weak-exclusive cache* that maintains private data exclusively and shared data inclusively.

Dead Block Prediction. Our architecture in Chapter 3 proposes predicting far-reuse blocks that will not be re-referenced before they are evicted (also called *dead blocks*). Since dead blocks will not be accessed again, eliminating such block from the cache hierarchies can significantly improve cache efficiency. To this end, there have been several approaches to predict dead blocks prior to its eviction.

Lai et al. [47] proposed trace-based dead block predictor and dead block correlating prefetcher. The predictor tracks a trace of memory access and predicts when a cache block is dead and evictable from L1 cache. The proposed prefetcher correlates the dead block with following memory access and replaces dead block with prefetched block which is likely to be referenced in near future. Khan et al. [48] introduced sampling dead block predictor which is similar to our reuse distance predictor. It predicts dead block based on the last instruction that accessed a block. The prediction result is used for replacement or bypass decision.

However, both techniques are incompatible with exclusive caches since they rely on access history, which assumes inclusive/non-inclusive caches. Moreover, those

techniques only predict dead blocks (i.e., far-reuse blocks) and are not able to identify near-reuse blocks, hence they can not take advantage of hybrid cache in our architecture.

2.3 Distributed Hybrid Cache

2.3.1 Prediction Hybrid Cache.

Prediction hybrid cache (PHC) [20] is a state-of-the-art hybrid cache architecture, which outperforms conventional migration-based hybrid caches. The key idea of PHC is to *predict* the write intensity of each cache block at the time of block load and use that information to guide block placement. For this purpose, it defines *cost* of a block as the analytic model of write intensity:

$$\begin{aligned} \text{cost} &= N_r \Delta E_r + N_w \Delta E_w \\ &= N_r \times (E_r^{STT} - E_r^S) + N_w \times (E_w^{STT} - E_w^S) \end{aligned} \tag{2.5}$$

In this equation, N_r and N_w are the number of reads and writes[1] while the block resides in the cache, and E_r^S (or E_r^{STT}) and E_w^S (or E_w^{STT}) are the read and write energy of SRAM (or STT-RAM), respectively. Therefore, the cost of a block implies the amount of extra energy consumption when the block is allocated into STT-RAM instead of SRAM. In other words, blocks with higher cost are more preferred to be allocated to SRAM from the energy efficiency perspective.

Based on this cost model, PHC predicts whether the cost of each block will exceed a write intensity threshold κ or not at the time of block insertion. It uses a PC-directed predictor that correlates a write-intensive block (i.e., cost $\geq \kappa$) and the instruction that triggers loading of the block. This enables highly accurate prediction of write intensity (e.g., 93% in the original article). In addition, since the amount of write-intensive data varies across different applications, PHC also has a mechanism that adjusts the write

[1] We slightly modified this cost model to include writes for block fills (on a cache miss, the loaded have has to be inserted into the cache, which incurs a write operation to the cache) into N_w.

intensity threshold κ at runtime. It tries to minimize the write energy consumption while not increasing the cache miss rate compared to the LRU policy (e.g., if too many blocks are allocated to SRAM, the STT-RAM capacity can be underutilized, which degrades the cache hit rate).

2.3.2 Distributed Cache Partitioning

In manycore systems, the increased number of cores in a chip allows plenty of applications to run simultaneously, thereby improving the system utilization and throughput. However, this creates a new challenge in that those applications interfere with each other for shared resources (e.g., LLCs), and thus results in the degradation of Quality-of-Service (QoS) and hampers the throughput improvement from manycore systems. For this problem happening at the LLC, cache partitioning [22, 23, 49–52] is one of the most popular solutions. It restricts the cache size that can be occupied by each application (called *partition*) to isolate the impact on LLC performance of each application within each partition. However, most of them are not scalable and only focus on centralized caches, and distributed cache partitioning for manycore systems [22, 23] are also proposed. In our work explained in Chapter 5, we focus on a state-of-the-art cache partitioning for distributed cache architectures, called *Jigsaw*.

Jigsaw [22] partitions distributed caches by introducing a new concept of virtual caches (called *shares*). Each application has its own share, which is constructed with multiple partitions (each of which is located at a single bank) from different banks and is determined by the placement policy. From the local bank perspective, each bank locally adopts a scalable cache partitioning scheme (e.g., Vantage [52]) to manage multiple partitions in a bank from different shares. Globally, the size and placement of each share are periodically updated in a way to minimize the total number of misses based on the monitored statistics. In this way, Jigsaw can address both scalability and interference issues in distributed caches at the same time.

Jigsaw also has a mechanism to enable efficient lookups in distributed cache architectures. It adds a special structure called *share-bank translation buffer (STB)*, which keeps the information about which cache blocks are stored in which banks. Since STB gives the exact location of the given cache block, it does not have to traverse multiple banks to find out where the cache block is stored, unlike other distributed cache architectures.

2.3.3 Related Work

Highly-Associative Caches. Since conventional set-associative caches directly use part of memory address as a set index, they suffer from uneven access distribution across the sets in the cache (explained in Section 5.1). In addition, since they select a replacement candidate within a set, blocks in frequently accessed sets are easier to be evicted than those in sets with infrequent accesses.

To mitigate this, highly-associative caches are proposed. The skewed-associative cache [53] utilizes hash functions to pick more even distribution of set indexes from memory addresses, which improves the set-level write distribution and the effective associativity of the cache. ZCache [54] is another highly-associative cache design that utilizes hash functions combined with cuckoo hashing to increase the number of replacement candidates. Thus, with only a small number of physical ways (e.g., 4 ways), ZCache outperforms set-associative caches with higher associativity (e.g., 32 ways). The V-Way cache [55] increases the number of sets in the tag array while maintaining the same data array size and maintains only the tag array as conventional set-associative caches with indirection from the tag array to the data array (called *forward pointer*). Since it uses more sets in the tag array, it reduces the chance of conflict misses.

Cache Partitioning. Cache partitioning is composed of an allocation policy and a partitioning scheme. The allocation policy decides the size of a partition to be allocated

to each application. Based on the partition size information, the partitioning scheme enforces each application to use the allocated portion of the shared cache. Utility-based cache partitioning [50] is a state-of-the-art allocation policy that decides the optimal size of each partition by estimating the number of misses in each application under all possible partition configurations and choosing the one with the fewest cache misses. The most popular partitioning scheme is way partitioning [49], which partitions the cache ways in a set according to the partition decision and enforces each application to use its own ways; however, way partitioning is not scalable for manycore systems because the number of partitions is limited by the number of physical ways.

In order to overcome the limitations of conventional partitioning in manycore systems, there have been several approaches to *scalable* cache partitioning supporting a large number of partitions (e.g., 64 in our configuration). Promotion/insertion pseudo-partitioning (PIPP) [51] supports fine-grained partitioning by modifying the cache replacement policy; however, it does not strictly guarantee the size and the isolation of each partition. Vantage [52] utilizes highly-associative caches [53–55] to support line-granularity partitioning based on a modified replacement policy. The key idea is to divide the cache into a main region (called *managed region*) and a victim cache (called *unmanaged region*) and control the rate of inserting cache blocks into each partition and demoting them to the unmanaged region to adjust the size of each partition. CloudCache [23] is another partitioning scheme for manycore systems. Similarly to Jigsaw, it also distributes virtual caches to multiple banks with way partitioning, but it broadcasts local cache misses to reduce the cache latency, which degrades its scalability. Our architecture uses Jigsaw (which internally uses Vantage as a partitioning scheme of each LLC slice) as the baseline, but our ideas can be generalized to other cache partitioning schemes for manycore systems.

Decoupling Tag Array and Data Array. In Chapter 5, our intra-bank optimization proposes a tag-to-data pointer table to decouple the tag array and the data array. There have been several approaches that use similar structures to enable a better cache replacement policy with many-to-one mapping from tags to data blocks [55, 56] or sharing of multiple data arrays across different nodes in a NUCA configuration [57, 58]. While our tag-to-data pointer table shares structural similarity with those prior approaches, our work is novel in that it leverages decoupling of the physical location of data blocks from that of tags (realized by the tag-to-data pointer table or similar structures) to *efficiently enable highly-associative hybrid caches,* namely eliminating relocation overhead and avoiding unintended cross-region migration (i.e., STT-RAM to SRAM or vice versa).

Chapter 3

Exclusive Last-Level Hybrid Cache

3.1 Motivation

3.1.1 Exclusive Cache Hierarchy

To maintain an exclusion property in exclusive cache, some cache operations behave differently from those of inclusive/non-inclusive caches. First, a cache block is inserted into the LLC *only after it is evicted from the upper-level caches*. In inclusive caches, a LLC miss loads the target block into both the LLC and the upper-level caches to maintain the inclusion property. On the contrary, exclusive caches bring cache blocks to the upper-level caches first and move them to the LLC on eviction to prevent duplication.

Second, every LLC block is invalidated from the cache right after its first hit to preserve the exclusion property. This is different from inclusive caches where cache blocks are rather duplicated (instead of being moved to the upper-level caches) on hit for inclusion property.

Although these two differences improve cache efficiency by avoiding duplication of cache blocks, we observed that they incur significantly higher write overhead compared

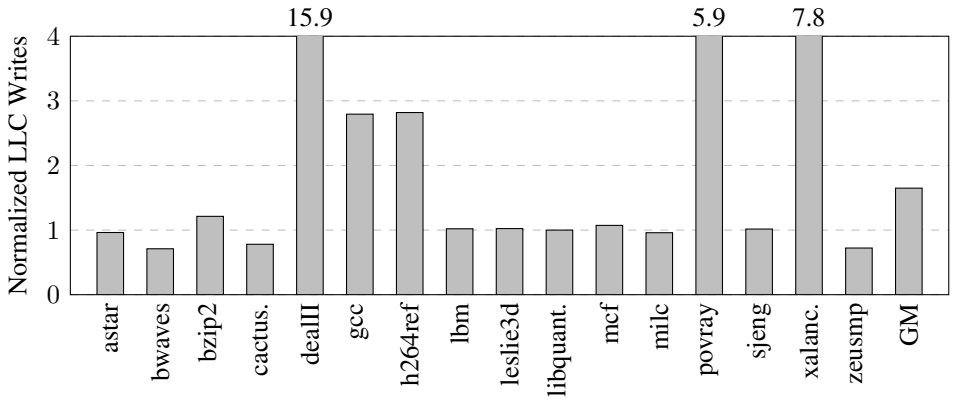


Figure 3.1 The number of LLC writes in the exclusive cache hierarchy normalized to that of the inclusive cache hierarchy.

to inclusive caches, which is problematic in STT-RAM caches due to their inefficient write operations. As Figure 3.1 shows, an exclusive LLC receives 65% higher average write traffic compared to its inclusive version (see Section 3.3.1 for evaluation methodology).

Unfortunately, previous work on STT-RAM cache write reduction [20,34,35,40–43] is not compatible with exclusive caches since they are designed based on cache behavior under inclusive/non-inclusive caches. For example, most of the existing inclusive hybrid cache architectures keep track of access information (e.g., the number of writes per block) and use it as a metric to perform block placement or migration for write energy reduction [20, 34, 35, 40]. This, however, does not work for exclusive caches, since they invalidate corresponding cache blocks on LLC hit (i.e., every block receives only one hit before its eviction) and therefore it is impossible to keep track of their access information. This motivates the need for an architectural technique *specifically for exclusive cache hierarchies* to alleviate the write overhead of STT-RAM LLCs.

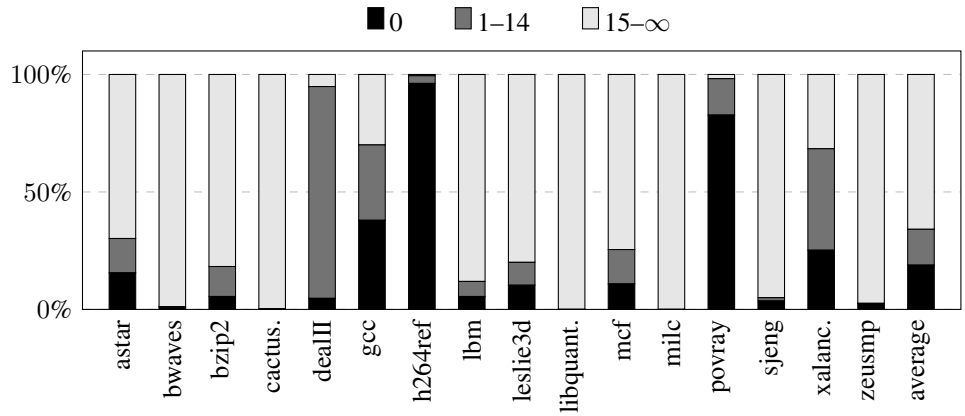


Figure 3.2 Reuse distance distribution.

3.1.2 Reuse Distance

Our key observation for mitigating the write overhead of exclusive caches is that *a significant portion of blocks inserted into exclusive LLCs are either (1) accessed in a very near future or (2) not accessed at all until their eviction*. In the first case (i.e., near reuse), since such blocks reside in the cache in a very short amount of time, it can be stored in a small, auxiliary buffer that has low write overhead (e.g., SRAM cache) to reduce write energy. In the second case (i.e., far reuse), such blocks do not need to be stored into any cache as they will not be re-referenced at all until their eviction. Thus we can avoid writes to an STT-RAM cache in those two cases.

As empirical evidence of such behavior, Figure 3.2 shows our experimental results on reuse distance profiling. *Reuse distance* of a target cache block is defined as the number of cache accesses to other blocks in the same set between two consecutive accesses to the target block. As can be seen in the figure, most of the cache blocks in the exclusive LLC exhibit either near reuse (reuse distance < 1, 19%) or far reuse (reuse distance > 14, 66%). This shows the potential of reuse distance as a metric for cache block classification for write overhead reduction in exclusive caches. In the following

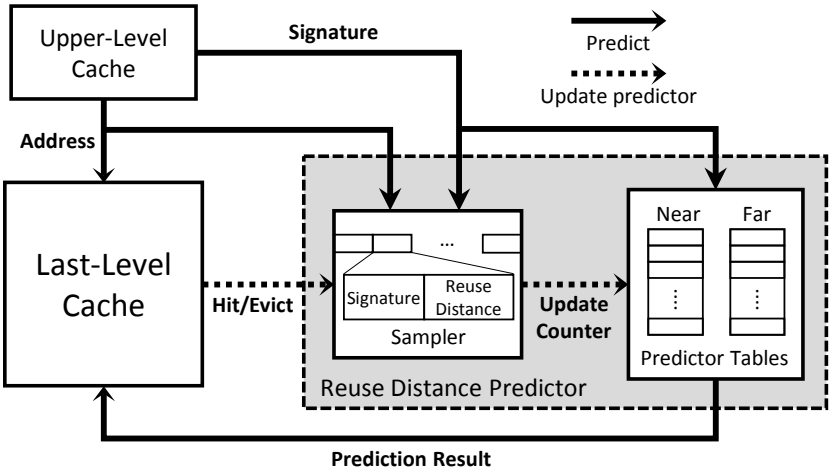


Figure 3.3 Overview of the proposed architecture.

section, we will describe our cache architecture that uses reuse distance prediction for energy-efficient exclusive hybrid cache management. Note that, unlike previous work on reuse distance prediction that focused only on improving performance of traditional SRAM caches [59, 60] and/or required software modifications [37, 61], our work borrows just the idea of reuse distance prediction and improves energy efficiency of exclusive hybrid caches with a hardware-based solution (i.e., no modifications to existing software).

3.2 Architecture

Figure 3.3 shows an overview of the proposed exclusive LLC architecture. The LLC is a hybrid cache combining a large STT-RAM region and a small SRAM region. This section explains the details of our hybrid cache management scheme.

3.2.1 Reuse Distance Predictor

Our reuse distance predictor shown in Figure 3.3 is designed based on the previous work called Signature-based Hit Predictor using program counters (PCs) as signatures

(SHiP-PC) [59]. To reduce the area overhead, it samples 1/32 of the entire cache sets[1] [62] and maintains signatures and reuse distances only for the sampled sets in a separate hardware called sampler. For each access to a sampled set in the LLC, the corresponding reuse distance information maintained in the sampler is updated (i.e., reuse distances of all blocks in the set except the one for the accessed block are incremented by one). The reuse distance field is set to zero on block insertion.

As a signature of the predictor, we associate each block with the PC of the instruction that has issued the access to it in the upper-level cache (i.e., the last access to the block when the block was in the upper-level cache before it is moved to the LLC). For this purpose, each upper-level cache tag is equipped with an additional field that stores the last-access PC, which is transferred to the predictor on its eviction. We extract 13 least significant bits from the PCs as signatures instead of using full 32-bit PCs since the tables (predictor tables, explained below) addressed by the PC are composed of 8192 entries (which requires 13 bits for indexing).[2]

For each signature value, the architecture tracks the reuse distance information records the trend in tables of 3-bit saturating counters (structurally similar to branch predictors). There are two predictor tables in our architecture, the near-reuse predictor table and the far-reuse predictor table. The former predicts if the next access with the given signature will have reuse distance shorter than a user-defined threshold Th_{near}, while the latter predicts if the next access will have reuse distance longer than a user-defined threshold Th_{far}.

The predictor tables are updated in two cases. First, when a cache block in the LLC is accessed, its reuse distance is compared with Th_{near} and Th_{far} to update the counters addressed by the signature of the current access. If the reuse distance of the

[1] According to our experiments, employing set sampling in our architecture increases LLC energy by a small amount (5.9% on average) compared to the one without sampling due to slightly more misprediction.

[2] We have also evaluated our architecture using 15 LSBs of PC (instead of 13 bits) and observed an ignorable improvement (only 0.3% more reduction of LLC energy consumption) at a cost of 4x larger tables.

block is shorter than Th_{near}, the corresponding counter in the near-reuse predictor table is incremented by one. Otherwise, the counter is decremented. Similarly, the counter in the far-reuse predictor table is incremented when the reuse distance is longer than Th_{far} and decremented otherwise. The second case of updating the counters is when a cache block is evicted from the LLC. In that case, the counters are updated as if the reuse distance of the evicted block were infinite and using the same mechanism described above.

Based on the counter values, the reuse distance prediction is made when the LLC receives a block insertion request. First, it accesses the two predictor tables and reads the counters corresponding to the PC from the request. According to the values of the near-reuse table counter (N) and the far-reuse table counter (F), we predict blocks with '$F \geq 4$' as far reuse distance and '$F < 4$ and $N \geq 4$' as near reuse distance (the value '4' is empirically determined). This information is used to determine block placement in our architecture, which will be discussed in the following section.

3.2.2 Hybrid Cache Architecture

Using predicted results from the reuse distance predictor, we develop a hybrid cache architecture for exclusive LLCs. Each cache set of our architecture is composed of many STT-RAM blocks and few SRAM blocks (e.g., 15 and 1 blocks per set, respectively, for a 16-way set-associative cache). This achieves better energy efficiency than homogeneous approaches since STT-RAM provides low static power, while SRAM shows much lower write energy and latency at the cost of increased static power.

When a cache block is evicted from the upper-level cache, our architecture needs to determine whether the block needs to bypass the LLC or to be inserted into the SRAM region or the STT-RAM region. For this purpose, we classify cache blocks into near, medium, and far reuse categories using reuse distance prediction and exploit such information to perform efficient block placement for our architecture as follows.

First, for cache blocks predicted as far reuse distance, we bypass the block insertions from the cache (and write them back to main memory if they are dirty). The key idea behind this is that inserting these blocks is wasteful since blocks with far reuse distance will be most likely evicted before they receive hit (if any). Since the misprediction penalty of bypassing could be high due to extra cache misses, if there is a free space in the STT-RAM region (i.e., at least one invalid block exists in the set; note that unlike inclusive caches, exclusive caches can have invalid blocks even after warm-up since blocks are invalidated on hit), we insert blocks into the region even when they have far reuse distance. Those blocks are marked with one extra bit per block in the tag array and are evicted with higher priority on block replacement.

Second, cache blocks predicted as near reuse distance are inserted into the SRAM region. The rationale of this is that those blocks are accessed very soon after their insertion, which makes the small SRAM region sufficient to hold them until they are accessed. Inserting near reuse blocks into the SRAM region is particularly beneficial in exclusive caches since blocks are invalidated on cache hit, which makes room for another block to be cached in the SRAM region, thereby making the best use of the small SRAM region. Also, if a cache block in the SRAM region is not accessed at all before its eviction, we insert it into the STT-RAM region at its eviction instead of evicting them from the LLC to give a second chance since those blocks are likely to receive hits according to the prediction results.

Third, all other cache blocks (medium reuse distance) are inserted into the STT-RAM region as they are expected to be accessed before their eviction, but not in the near future. Through this, short-lived blocks exploit small write overhead of SRAM, while only long-lived ones pay high write cost of STT-RAM to benefit from its low static power.

3.3 Evaluation

3.3.1 Methodology

Our architecture is evaluated by using MacSim [63], a trace-driven, cycle-level x86 simulator. The baseline system has a four-issue, out-of-order core with 256 reorder buffers operating at $4\,\mathrm{GHz}$ and $32\,\mathrm{KB}$ 8-way set-associative L1 instruction/data caches having 64-byte blocks, which use the LRU replacement policy. We use DDR3-1600 timing parameters with configuring two channels and eight banks per channel for the main memory of the system.

We use a $1\,\mathrm{MB}$ 16-way set-associative LLC with 64-byte blocks using the NRF (not recently filled) replacement policy [64] as the LLC of the baseline system. In the proposed hybrid LLC, each cache set is composed of one SRAM block and fifteen STT-RAM blocks (the numbers of ways are determined empirically) to minimize the high static energy consumption of the SRAM region.

For the reuse distance predictor, we use two 8192-entry predictor tables with 3-bit saturating counters. For reuse distance classification, we set Th_{near} to the number of SRAM ways in the hybrid cache (i.e., 1 in our current implementation) and Th_{far} to infinity to consider only the blocks that are evicted without reuse as having far reuse distance. Dynamic adjustment of the threshold according to application characteristics is part of our future work.

Table 3.1 shows characteristics of SRAM/STT-RAM used in our evaluation, which are modeled by CACTI [65] and NVSim [66], respectively, under the 45nm technology. We use LOP (Low Operating Power) cells for peripheral circuits of tag/data arrays. We refer to the previous work for the cell characteristics of STT-RAM [67, 68]. We also model energy overhead of the reuse distance predictor in our architecture, which is negligible compared to the LLC energy (5.0% on average as shown in Figure 3.5). Since the reuse distance predictor is not on the critical path, they do not affect the

Table 3.1 Characteristics of the SRAM, STT-RAM, and Hybrid Cache

	SRAM	STT-RAM	Hybrid
Read Latency (cycles)	12	12	12/12*
Write Latency (cycles)	12	44	12/44*
Read Energy (nJ)	0.04	0.10	0.04/0.10*
Write Energy (nJ)	0.04	0.63	0.04/0.63*
Static Power (mW)	16.10	5.02	5.71

* Values for SRAM/STT-RAM, respectively.

performance.

As workloads for the evaluation, we choose 16 write-intensive benchmarks[3] from SPEC CPU2006 that have high L2 writes per kilo instruction (WPKI) in the baseline system. For benchmarks with multiple input sets, we choose the representative ones based on the previous work [69]. All benchmarks are run for 500 million instructions of the representative phases extracted by PinPoints [70].

3.3.2 LLC Energy Consumption

Figure 3.4 compares the energy consumption of the LLC in the proposed architecture against the STT-RAM baseline. The baseline uses a monolithic STT-RAM cache without any improvement technique as a LLC (STT-RAM-Baseline). We show the effectiveness of our architecture by applying our bypassing scheme only (STT-RAM-Bypass) and then adopting our hybrid cache architecture on top of it (Hybrid-Bypass). We also include the results obtained by simply replacing STT-RAM with SRAM under our bypassing scheme (SRAM-Bypass) for comparison with the hybrid cache. Note that it is infeasible to compare with previously proposed hybrid caches since ours is the first hybrid cache architecture that can be applied to exclusive LLCs.

First, according to the evaluation results, our bypassing scheme alone reduces the

[3] Although not shown in this dissertation, our architecture consumes 11% less LLC energy compared to the STT-RAM baseline in 13 other benchmarks.

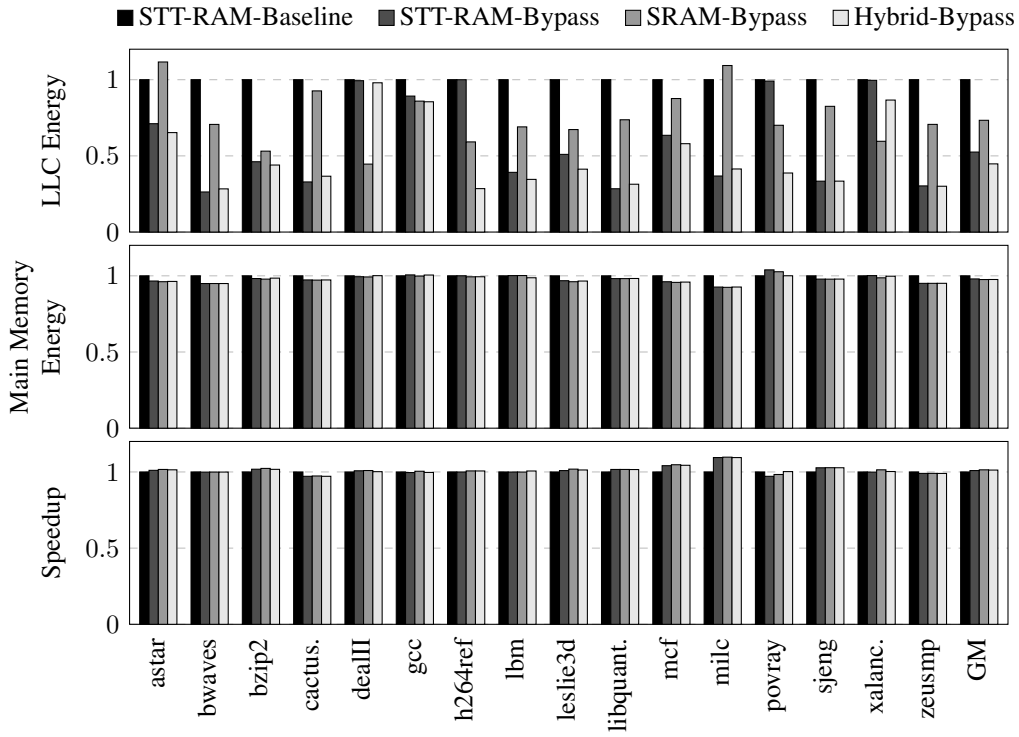


Figure 3.4 Energy consumption and performance of our architecture normalized to the STT-RAM baseline.

LLC energy by 48%. Through our scheme, most of the block insertions bypass the LLC, which reduces the write energy consumption by 75% on average. This is particularly effective in bwaves, cactusADM, libquantum, milc, sjeng, and zeusmp where most of the accesses belong to the far reuse distance category as shown in Figure 3.2.

Moreover, using SRAM/STT-RAM hybrid cache further reduces the LLC energy by 15% (55% from STT-RAM-Baseline). On average, the SRAM region absorbs 18% of cache writes to avoid costly STT-RAM writes for near reuse blocks. This allows us to further improve the energy efficiency of our scheme in benchmarks such as h264ref, povray, and xalancbmk where the bypassing scheme is ineffective. As can be seen in Figure 3.2, such benchmarks have a considerable amount of blocks with near reuse distance, which should not bypass the cache but can be well served by a small SRAM cache (e.g., 98% of cache writes are serviced by SRAM in h264ref).

In addition, our hybrid cache architecture also consumes less cache energy compared to the one that uses SRAM only (SRAM-Bypass). Simply replacing the entire STT-RAM with SRAM leads to a noticeable increase in LLC energy due to the higher static power consumption. Since such behavior highly depends on the write intensity of applications, there is no clear winner between STT-RAM-Bypass and SRAM-Bypass in terms of energy efficiency. On the contrary, our architecture substitutes only a small portion of the STT-RAM cache with SRAM and employs a mechanism that fully utilizes it thereby facilitating adaptation to application characteristics.

However, some benchmarks show a slight increase in LLC energy after applying the hybrid cache technique (e.g., bwaves, cactusADM, libquantum, and milc). This is because they do not have an enough amount of writes with near reuse distance to offset the increased static power of SRAM by exploiting its low write energy. This can be confirmed by the observation that far reuse distance dominates the cache accesses in those benchmarks as shown in Figure 3.2. Also, dealII benefits from neither bypassing nor hybridizing since most of the cache accesses in dealII belong to medium reuse as

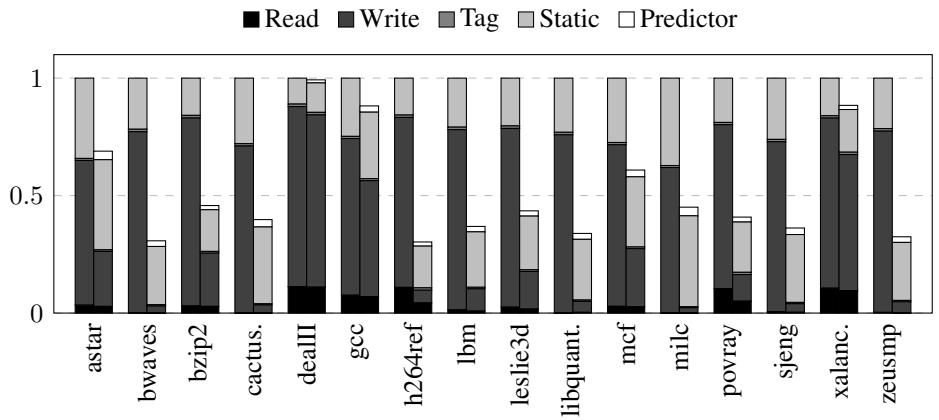


Figure 3.5 Energy breakdown of STT-RAM-Baseline (left) and Hybrid-Bypass (right).

shown in Figure 3.2.

In summary, Figure 3.5 compares the energy breakdown of our architecture against the STT-RAM baseline. Most noticeably, LLC energy consumption in STT-RAM-Baseline is dominated by the high write energy of STT-RAM. In contrast, our architecture greatly reduces the write energy consumption of exclusive LLCs at the cost of a modest increase in static power and negligible overhead from the reuse distance predictor.

3.3.3 Main Memory Energy Consumption

Since both bypassing blocks and using hybrid caches may affect the cache miss rate, we also measure the LLC miss rates across different configurations. On average, Hybrid-Bypass reduces the LLC miss rate by 0.53% compared to STT-RAM-Baseline because bypassing improves cache efficiency by not inserting cache blocks with no reuse. This saves main memory energy consumption by reducing main memory accesses. In addition, slight performance improvement (see Section 3.3.4) further reduces energy consumption of main memory. Accordingly, as shown in Figure 3.4, our analysis on main memory energy consumption using Micron's Power Calculator [71] indicates that

Hybrid-Bypass consumes less energy (2.4% on average) in the main memory compared to STT-RAM-Baseline.

3.3.4 Performance

Figure 3.4 also shows the performance of our architecture in terms of instructions per cycle (IPC) normalized to that of the STT-RAM-Baseline. We observed that STT-RAM-Bypass and Hybrid-Bypass slightly improve the performance by 0.9% and 1.2%, respectively, compared to STT-RAM-Baseline. The performance improvement of our architecture is mainly due to the reduction in bank contention caused by long write operation and miss rate mentioned in the previous section.

3.3.5 Area Overhead

Our architecture incurs very small area overhead. In the LLC, we add one bit per block to record the replacement priority for blocks predicted as far reuse distance (see Section 3.2.2). The reuse distance predictor is comprised of two predictor tables with 8192 3-bit counters and a sampler that stores a 13-bit signature and a 1-bit reuse distance[4] of each LLC block in the sampling sets (32 sets out of 1024 sets in our environment). Lastly, each L1 data cache tag is extended with a 13-bit last-access signature field for reuse distance prediction. In total, storage overhead of our technique is only $0.8\,\mathrm{KB}/8.9\,\mathrm{KB}$ (2.5%/0.9%) in the L1/L2 cache.

3.4 Summary

In this work, we proposed an energy-efficient cache architecture to mitigate write overhead when combining STT-RAM with exclusive caches. Although the exclusive caches are able to utilize cache capacity more efficiently over inclusive caches, it may increase write operations of caches. To reduce harmful STT-RAM writes, the

[4]We need to track reuse distances up to Th_{near} (i.e., 1), which requires one bit for each field, since Th_{far} is set to infinity.

architecture takes a hybrid cache consisting of SRAM and STT-RAM as LLC. Based on the reuse distance prediction, a cache block is placed into SRAM or STT-RAM or bypass the LLC.

Chapter 4

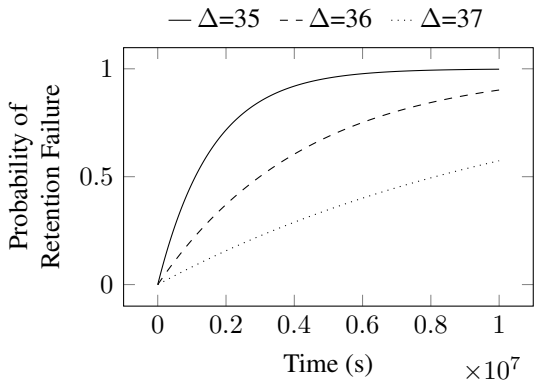
Designing Volatile STT-RAM Cache

4.1 Analysis

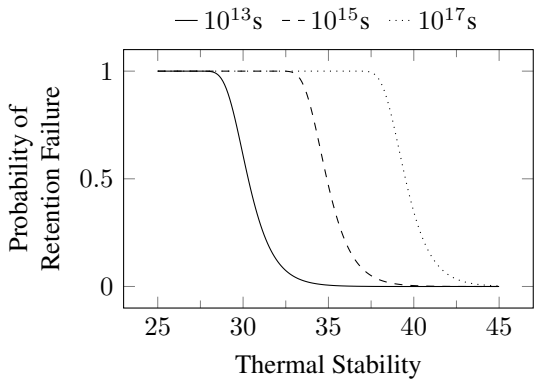
Based on the analytic models explained in the Section 2.1, this section provides detailed analysis of a volatile STT-RAM cell and trade-offs when designing an array of volatile STT-RAM.

4.1.1 Retention Failure of a Volatile STT-RAM Cell

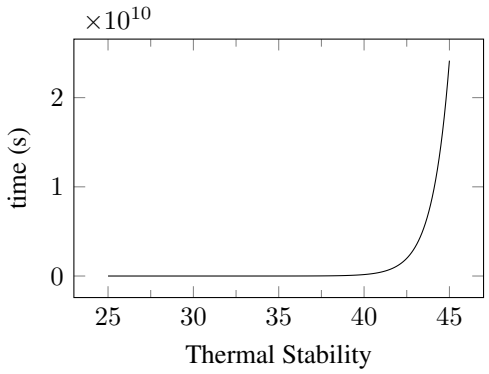
Figure 4.1 shows retention failure characteristics of an STT-RAM cell drawn based on the aforementioned models. First, Figure 4.1a illustrates the probability of retention failure rate as a function of elapsed time while thermal stability is fixed. As expected, the probability of failure increases as time flows and increases rapidly right after the data is written. However, it takes very long time to reach a high probability value (e.g., probability becomes 0.5 after 34.6 days when Δ=36). Secondly, Figure 4.1b shows the probability of retention failure on various STT-RAM cells having different thermal stability for fixed elapsed times. From this figure, we can discover that the retention failure can be significantly reduced with only a small change in thermal



(a)



(b)



(c)

Figure 4.1 Retention failure characteristics of an STT-RAM cell.

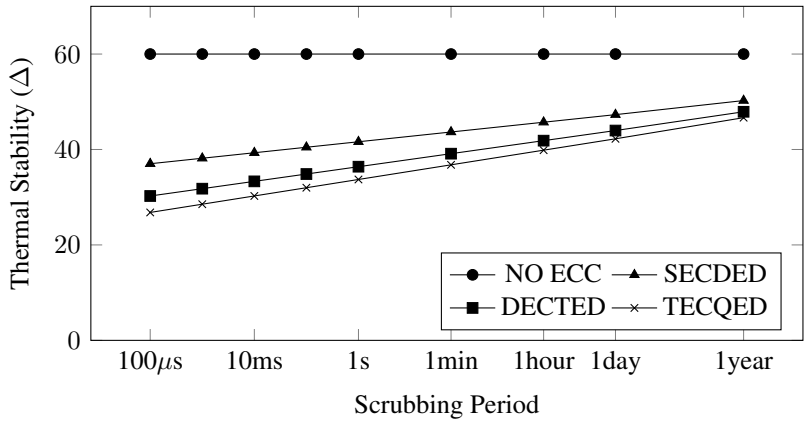


Figure 4.2 Relationship between memory array design parameters.

stability. Thirdly, Figure 4.1c shows required time elapse to reach a specific probability of retention failure (0.5 in this example) on different thermal stability. The required time increases very sharply as thermal stability increases. From this, we can conclude that the retention failure rate of STT-RAM is more sensitive to thermal stability than time; slightly higher thermal stability increases retention time significantly.

4.1.2 Memory Array Design

Figure 4.2 summarizes the relationship between the parameters of designing memory array with volatile STT-RAM. It shows the minimum thermal stability obtained by (2.4) when the scrubbing period varies for different ECC strengths. No ECC implies non-volatile STT-RAM, which is not scrubbed at all. As scrubbing period increases exponentially, minimum thermal stability increases linearly and very slowly. On the other hand, using ECC significantly reduces minimum required thermal stability compared to no ECC (non-volatile STT-RAM) and the thermal stability gap between two different ECC strengths decreases as ECC strength grows.

From this figure, three important information can be extracted. First, frequent scrubbing might be very inefficient since it reduces the required thermal stability only

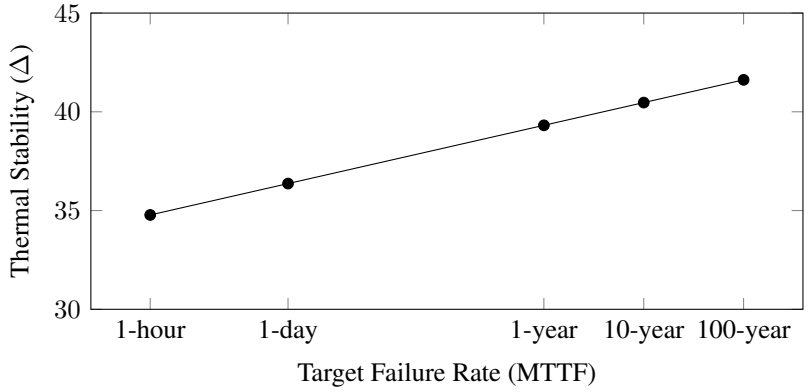


Figure 4.3 Relationship between target failure rate and thermal stability.

slightly while significantly increasing scrubbing overhead in terms of performance and energy consumption. Thus for energy efficient memory, scrubbing period should be set long enough for negligible overhead even if it incurs slight increment in thermal stability. Secondly, using single error correction double error detection (SECDED) ECC is very effective to reduce thermal stability while using stronger ECC (DECTED and TECQED[1]) is less effective when considering the increased ECC overhead. Thirdly, although a considerable increment of scrubbing period incurs only a slight increment of thermal stability, extremely long scrubbing period like 1 year may suffer from increased thermal stability. Moreover, in such a case, even if a stronger ECC is used, the thermal stability does not decrease much.

Relationship between target failure rate of a volatile STT-RAM cache array and thermal stability is shown in Figure 4.3. As can be seen in the figure, the trend of thermal stability when changing target failure rate is similar to that of changing scrubbing period. As the target failure rate decreases (longer MTTF in Figure 4.3)[2] exponentially, the

[1]DECTED and TECQED stand for double error correction triple error detection and triple error correction quadruple error detection, respectively

[2]Longer MTTF (Mean Time To Failure) means that an error is expected to occur in longer period. Thus, longer MTTF means lower failure rate.

thermal stability increases linearly and very slowly.

Summarizing the discussion, in volatile STT-RAM array design, short scrubbing period and strong ECC may be the best choice for low thermal stability. For efficiency, however, scrubbing period and ECC strength should be carefully determined considering their overhead.

4.2 Evaluation

4.2.1 Methodology

For evaluation, we use MacSim [63], a trace-driven, cycle-level x86 simulator. The system runs at 4GHz with four-issue, out-of-order core having 256 reorder buffers and has 32 KB 8-way set-associative L1 instruction/data caches with 64-byte blocks, which use the LRU replacement policy. As an L2 cache (LLC), we use 2 MB 16-way set-associative cache with 64-byte blocks which is configured with STT-RAM of various thermal stability. LLC access and scrubbing operations are interleaved among four banks. Scrubbing is performed only to valid blocks and the scrubbing operations are spread over every scrubbing period to avoid crowded cache operations and thus have less bank contention. For the main memory system, DDR3-1600 timing parameter with two channels and eight banks per channel are used.

STT-RAM caches are modeled by NVSim [66] under 45nm technology using LOP (Low Operating Power) cells for peripheral circuits. We use the same STT-RAM cell characteristics in the previous work [14, 67, 68, 72].

Extra overhead for ECC encoding/decoding and storing extra bits is also considered. The characteristics of the ECC encoders/decoders are extracted from the synthesized results obtained by using the Synopsys Design Compiler and summarized in Table 4.1. BCH based ECC is used in this work.

As workloads for the evaluation, SPEC CPU2006 is used. For benchmarks with multiple input sets, we choose one representative input set based on the previous

Table 4.1 Characteristics of ECC encoder/decoder

		SECDED	DECTED	TECQED
Encoder	latency	2 cycles	2 cycles	2 cycles
	Dynamic Energy	0.06 pJ	0.10 pJ	0.14 pJ
	Static Power	32.55 μW	46.12 μW	59.83 μW
Decoder	latency	4 cycles	8 cycles	12 cycles
	Dynamic Energy	6.62 pJ	16.19 pJ	19.10 pJ
	Static Power	1.74 μW	3.82 μW	5.66 μW

work [69]. For each benchmark, we run 500 million instructions of representative phases extracted by PinPoints [70].

For the baseline system, conventional non-volatile STT-RAM is used without ECC and scrubbing. We evaluated volatile STT-RAM with nine different scrubbing periods (100μs, 1ms, 10ms, 100ms, 1s, 1min, 1hour, 1day, 1year), three different error correcting capacities (SECDED, DECTED, TECQED), and five different target failure rates (1-hour, 1-day, 1-year, 10-year, 100-year MTTF). Figure 4.2 and 4.3 show minimum thermal stability of each configuration. Thermal stability of non-volatile STT-RAM is set to 60 [10]. To evaluate the impact of ECC strength and scrubbing period, the target failure rate is set to 11415 FIT (failures in time; the number of failures in one billion hours) which corresponds to 10-year MTTF. For the evaluation of different target failure rates, ECC strength and scrubbing period are fixed to SECDED and 100ms, respectively, since that configuration gives the best result as shown in the next subsection.

4.2.2 Last-Level Cache Energy

Figure 4.4 shows the energy consumption of the LLC on different scrubbing period and ECC normalized to the conventional non-volatile STT-RAM LLC. Each bar shows the average energy consumption of all SPEC2006 benchmarks for each configuration.

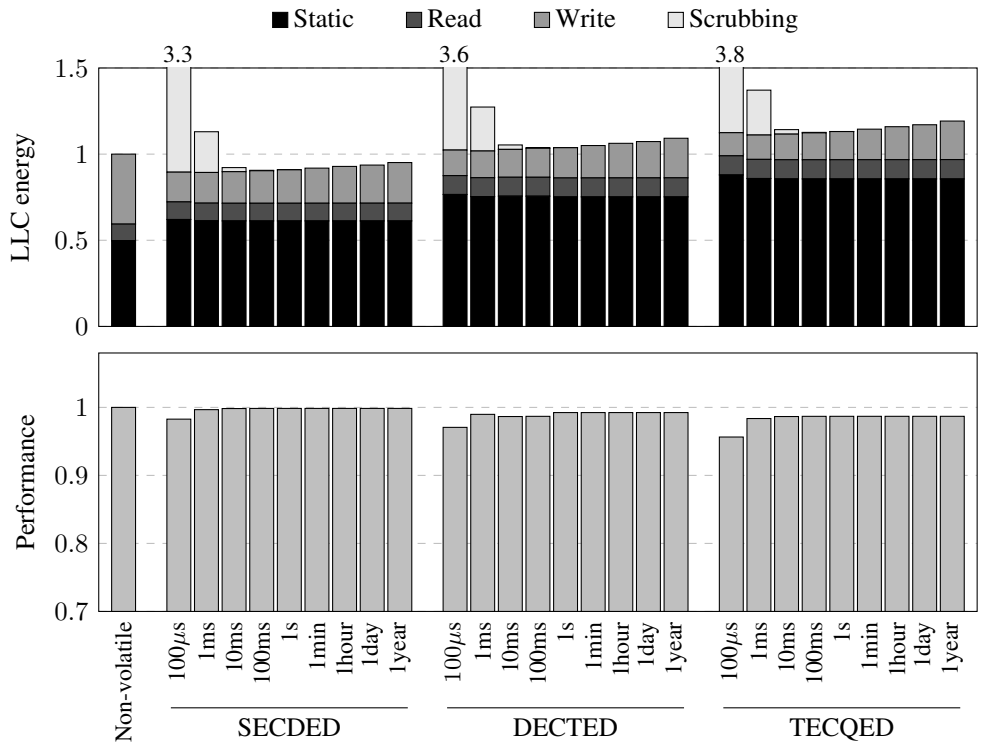


Figure 4.4 Last-level cache energy consumption and performance of various configurations.

Figure 4.5 also shows the energy consumption of the LLC with different target failure rate and also normalized to the conventional non-volatile STT-RAM LLC. For each configuration, the LLC consists of STT-RAM with a different thermal stability as illustrated in Figure 4.2 and 4.3. As mentioned above, the static and dynamic energy for the ECC encoder/decoder circuit and extra bits for the encoded ECC are included. This section analyzes the sensitivity of each design parameter for volatile STT-RAM LLC to energy consumption.

Scrubbing Period. As shown in Figure 4.4, scrubbing overhead is decreased as scrubbing period increases. With a very short scrubbing period like 100μs, the scrubbing significantly increases the overall energy consumption of the LLC. However, the overhead decreases as the scrubbing period increases and becomes negligibly small if the period is over 10ms (less than 2.4% when scrubbing period is 10ms and SECDED is used). On the contrary, to increase the scrubbing period, the thermal stability should be increased and thus write energy of the LLC increases. In general, however, compared to scrubbing overhead, the increase of the write energy due to the increase of the scrubbing period is much smaller since the thermal stability need to be increased only by a small amount (See Figure 4.2). In other words, designing a memory array with a very short scrubbing period like 100μs is very inefficient because of the significant scrubbing overhead while the reduction of thermal stability thus achieved cannot reduce the LLC energy that much. However, although the scrubbing overhead is decreased and become negligible when the period is longer than 10ms, write energy of the LLC increases due to the increased thermal stability and thus total LLC energy also increases. Overall, finding out an optimal scrubbing period (100ms in our evaluation) is important for LLC energy efficiency.

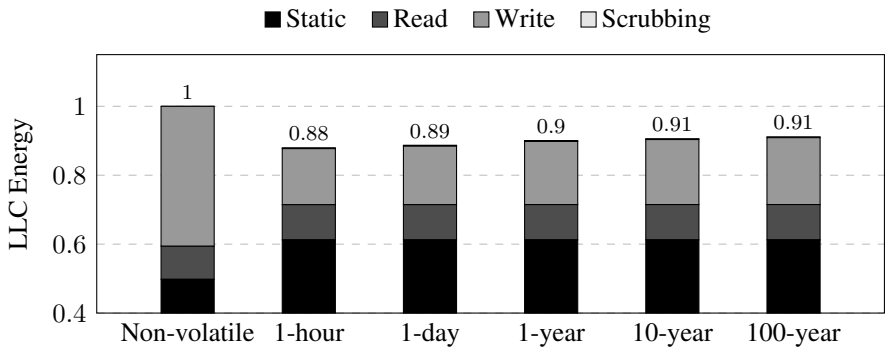


Figure 4.5 Last-level cache energy consumption on different target failure rates.

ECC Strength. For the same scrubbing period, LLC energy is increased as ECC strength is increased. Although the thermal stability of STT-RAM can be reduced with stronger ECC, the overhead of the encoder, decoder, and extra bits for the ECC is increased and the overhead grows faster than the energy saving from low thermal stability. Thus, using stronger ECC is not beneficial in terms of energy efficiency at all and it even consumes more energy over conventional non-volatile STT-RAM cache. As shown in Figure 4.4, SECDED gives the best result in terms of LLC energy and performance.

Target Failure Rate. Figure 4.5 shows an evaluation result of LLC energy consumption for different target failure rates at a fixed scrubbing period (100ms) and ECC strength (SECDED) which shows the best result as mentioned in the previous section. By allowing more errors, i.e., allowing higher target failure rate, the thermal stability of the STT-RAM cells can be decreased. As shown in the figure, loosening the target failure rate reduces LLC write energy. However, the amount of energy reduction is very small compared to the increase of the target failure rate (reducing LLC energy by 3% decreases the target MTTF from 100 years down to one hour).

Compared to non-volatile STT-RAM baseline, among various evaluated config-

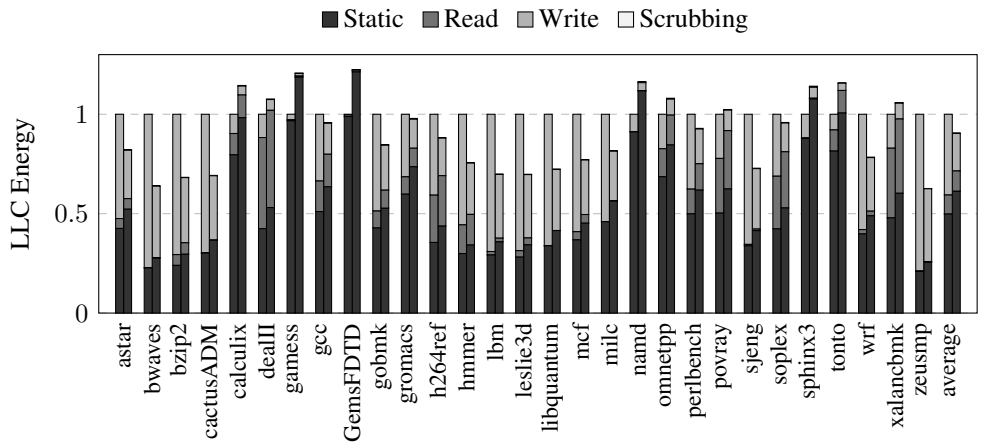


Figure 4.6 Last-level cache energy consumption for SECDED_100ms normalized to non-volatile STT-RAM baseline.

urations, only using SECDED with scrubbing period longer than 10ms saves LLC energy. The best configuration that uses SECDED with scrubbing period of 100ms (SECDED_100ms) consumes 9.4% less average energy than the baseline. Further decreasing the scrubbing period or using a stronger ECC consumes more LLC energy because the overhead of scrubbing or ECC is larger than the energy saving obtained by lowering thermal stability. In particular, 100μs scrubbing period is very inefficient because of the frequent scrubbing, which consumes more than 3x LLC energy compared to the baseline. Using a very long scrubbing period can eliminate the scrubbing overhead but is not the best option because of increased write energy (e.g., SECDED_1year consumes about 5% more LLC energy than SECDED_100ms).

Figure 4.6 compares LLC energy consumption of the baseline and SECDED_100ms, which is the most energy efficient configuration on average, for all SPEC2006 benchmarks. On average, SECDED_100ms saves LLC energy by 9.4% compared to the baseline. However, in some benchmarks (calculix, dealII, gamess, GemsFDTD, namd, omnetpp, povray, sphinx3, tonto, xalancbmk), it consumes more energy than the baseline by up to 22%. This is because these benchmarks are either non memory intensive

(calculix, gamess, GemsFDTD, namd, omnetpp, sphinx3, tonto) or read intensive (dealII, povray, xalancbmk). Note that the main benefit of volatile STT-RAM comes from reduced write energy, and thus, if there is not much saving in write energy, then the total LLC energy actually increases because the ECC circuit increases static and read energy (23% and 6% on average, respectively, for those benchmarks) of volatile STT-RAM. On the other hand, most of the benchmarks that reduce LLC energy are write intensive. Since the main advantage of volatile STT-RAM over non-volatile STT-RAM is low write energy, volatile STT-RAM can best exploit the advantage when the benchmark is write intensive.

4.2.3 Performance

Figure 4.4 also compares performance of various configurations and that of the baseline. Using volatile STT-RAM as the LLC does not improve system performance at all and it even degrades performance. There are two main factors of performance degradation. First one is scrubbing overhead. As can be seen in Figure 4.4, performance is degraded more with shorter scrubbing period. Since every scrubbing operation reads data/ECC and checks whether the data is correct or not, it may incur a number of bank contentions if scrubbing period is very short. Thus longer scrubbing period shows better performance over short period. Second, extra cycles for decoding ECC may cause performance degradation. These extra cycles make read access latency longer and may cause more bank contentions. Furthermore, stronger ECC requires longer decoding time (Table 4.1) hence it makes system slower. Therefore, the performance degrades with shorter scrubbing period and/or stronger ECC. With longer scrubbing period, performance degradation is reduced but it is still lower than the baseline due to the ECC decoding overhead and the gap is larger with a stronger ECC (SECDED_1year, DECTED_1year, and TECQED_1year degrade performance by 0.1%, 0.8%, and 1.3%, respectively). The worst configuration in our evaluation, TECQED with 100μs scrub-

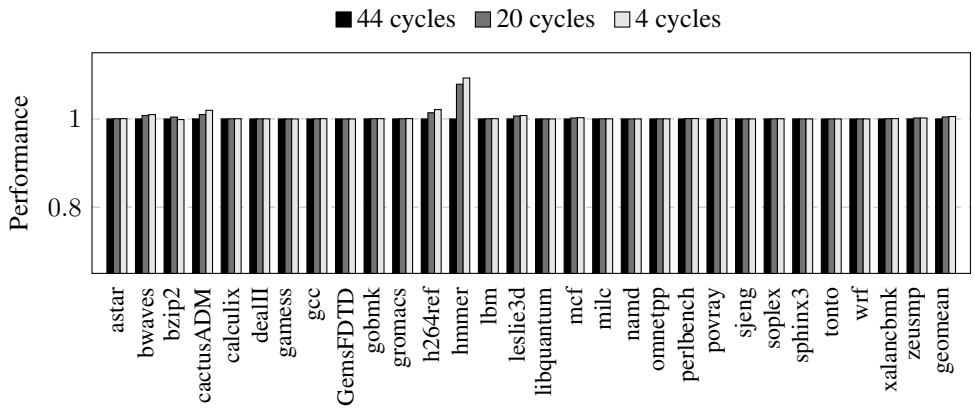


Figure 4.7 Performance for different write latency of STT-RAM.

bing period (TECQED_100μs), degrades performance by 4.4%. SECDED_100ms which is the best configuration in our evaluation also slightly degrades performance 0.2% compared to the baseline.

For different target failure rates with SECDED_100ms, a negligible performance drop (0.2% maximum) is observed since weak ECC (SECDED) and relatively long scrubbing period (100ms) have negligible overhead.

One of the advantages of volatile STT-RAM over non-volatile STT-RAM is shorter write latency. However, it does not affect much the overall performance because the write operations are in general not on the critical path. Figure 4.7 shows an empirical evidence for this statement. When the STT-RAM write latency changes from 44 cycles (longest write latency required for non-volatile STT-RAM) down to 4 cycles (shortest write latency achieved by TECQED_100μs), system performance varies by only 0.5% on average (up to 9.2%).

4.3 Summary

In this work, we explored trade-offs in the design of volatile STT-RAM. Relaxing the non-volatility of STT-RAM with lower thermal stability increases probability of

retention failure and thus periodic scrubbing with ECC is required to keep data securely. We first analyzed minimum thermal stability when the scrubbing period and the target failure rate is given. Then, based on the analysis, we evaluated volatile STT-RAM LLC with various configuration. The evaluation results show that weak ECC (SECDED) with reasonable scurbbing period (1ms) reduces energy consumption at most and provide a direction for designing volatile STT-RAM cache.

Chapter 5

Distributed Hybrid Cache

5.1 Motivation

The motivation in this work comes from our observation of write imbalance in distributed hybrid caches. As explained in Section 2.1.5, the effectiveness of a hybrid cache is determined by how efficiently the small SRAM can be utilized. Thus, to maximize the SRAM utilization, all writes have to be equally distributed across the entire cache, which, however, is not the case in conventional hybrid caches combined with distributed cache architectures.

First, conventional set-associative caches, on which existing hybrid cache architectures are based, experience significant write imbalance across sets (i.e., *intra-bank write imbalance*). This is because set indexes are determined only by the lower-order bits of the address, which incurs skewed access distribution particularly for applications with irregular memory access patterns. Considering that existing hybrid caches are built on set-associative caches with only a few SRAM ways in each set, such write imbalance can significantly degrade the SRAM utilization as some sets need to handle

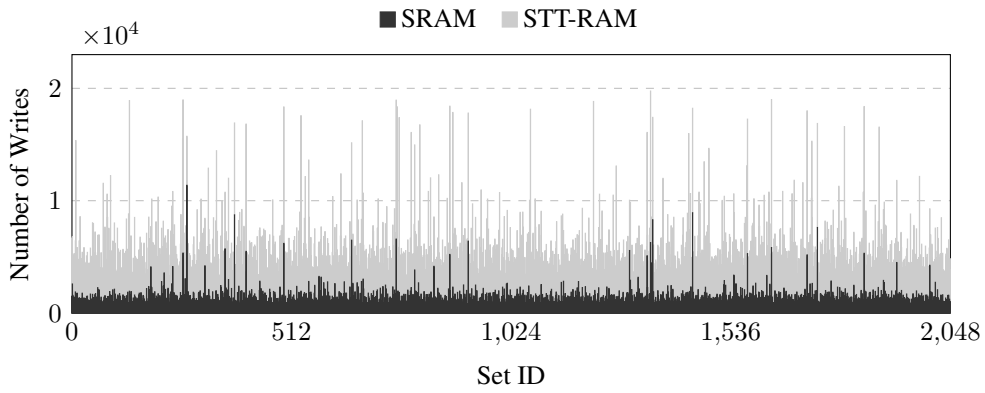


Figure 5.1 The distribution of SRAM/STT-RAM writes across different sets in a set-associative cache.

much more writes with the same number of SRAM blocks in a set. Figure 5.1 is an empirical evidence of such set-level write imbalance and low SRAM utilization. It shows the number of SRAM and STT-RAM writes in a prediction hybrid cache having one SRAM way and seven STT-RAM ways, measured by running mix of 64 applications from SPEC CPU2006 (similar to the configuration in ZCache [54]). As can be seen in the figure, the number of writes varies significantly across different sets, thereby incurring low SRAM utilization in some sets. For example, some sets (SRAM and STT-RAM together) receive up to 8.66 times as many writes compared to those with fewer writes. This leads to over a 22x difference in the number of SRAM writes in those sets (e.g., the set with the lowest SRAM utilization handles only 4.3% of writes in SRAM despite the 1:7 ratio of SRAM and STT-RAM ways). The dual-associative hybrid cache (DAHYC) [38] tries to address this inefficiency by allowing neighbor sets to share their SRAM blocks, but such a restricted style of sharing is expected to have a limited impact on mitigating the write imbalance since sets that receive more writes are often far apart from those with fewer writes.

Second, distributed caches introduce another level of write imbalance, which we call *inter-bank write imbalance*. As a motivational result, Figure 5.2 shows the distribution of

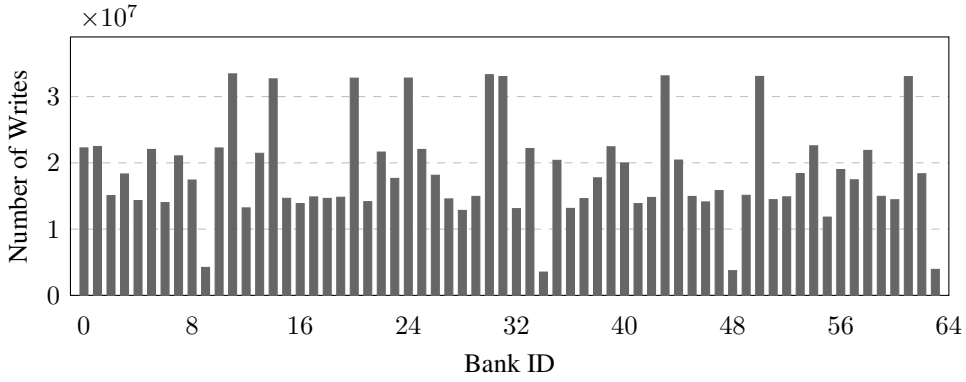


Figure 5.2 The distribution of writes across different banks in a distributed cache architecture (i.e., Jigsaw).

writes across different banks in Jigsaw (see Section 5.3 for system configuration), which demonstrates severe write imbalance across different banks (up to 9.5x difference). Such inter-bank write imbalance is mainly because manycore systems simultaneously run a large number of applications with different memory access patterns and each cache bank is shared by its own combination of application mixes with varying write intensity across different banks. This causes another dimension of inefficiency in distributed hybrid caches because SRAM in banks with fewer writes will be underutilized compared to those with more frequent writes.

Motivated by these two observations, our architecture aims at *evenly distributing write requests across different sets and banks in a distributed hybrid cache*. Globally balancing the write distribution facilitates higher SRAM utilization in distributed hybrid caches, thereby improving the energy efficiency. As will be explained in the next section, we address both intra-bank and inter-bank write imbalance problems through new architectural techniques.

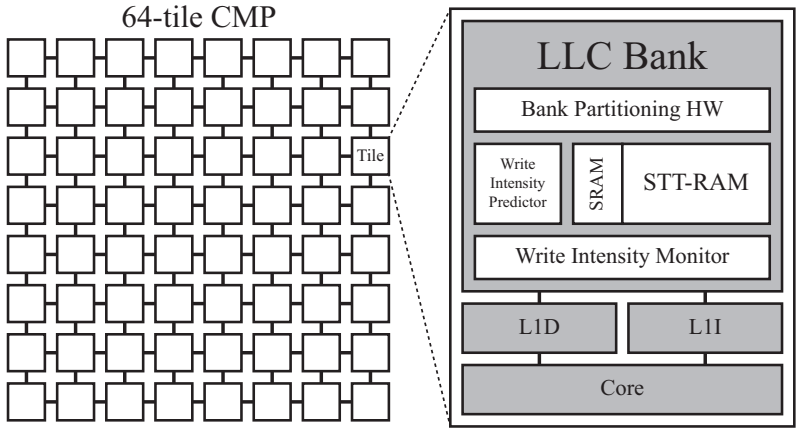


Figure 5.3 Overall architecture.

5.2 Architecture

We consider the tiled architecture used in Jigsaw [22] as a baseline of Benzene. As shown in Figure 5.3, each tile has a core, its own private cache (L1), and a slice of the shared LLC. The tiles are connected through an on-chip network.

On top of the baseline, Benzene applies PHC to each LLC slice and improves the efficiency of distributed hybrid caches at two different levels—intra-bank and inter-bank. The following gives an overview of our schemes. Note that, although we assume Jigsaw and PHC as concrete examples of underlying technologies for our architecture, our techniques can be generalized to other distributed cache partitioning and hybrid cache schemes as well.

Intra-Bank Optimization (Section 5.2.1). To balance the amount of writes to different cache sets, Benzene applies the concept of highly-associative caches [53–55] to hybrid caches and utilizes the idea of an indirection table [55, 57, 58], called *tag-to-data pointer table*, to minimize the energy overhead of using highly-associative caches in hybrid caches. With this design, the scarce SRAM resource in each bank can be shared

almost uniformly across all sets in a bank, thereby improving the energy efficiency of hybrid caches at the per-bank level.

Inter-Bank Optimization (Section 5.2.2). To reduce the write imbalance across all cache banks, Benzene tries to allocate data from write-intensive applications and non-write-intensive applications together in the same cache bank so that write-intensive data can be equally distributed to all banks, thereby enabling more efficient use of SRAM in each bank. To identify the write intensity of each application, a write intensity monitor (shown in Figure 5.3) collects the write statistics of each application and helps data placement. In addition, we devise an adaptive scheme that disables inter-bank optimization for non-write-intensive workloads to minimize its performance overhead.

Other Optimizations (Section 5.2.3). We introduce two modifications to the original PHC to be more suitable for distributed cache architectures: (1) set sampling that is aware of cache partitioning and (2) an improved threshold adjustment mechanism for escaping local optimum.

5.2.1 Intra-Bank Optimization

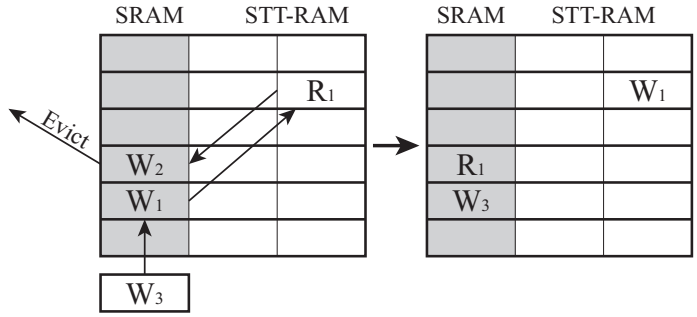
In order to mitigate the intra-bank write imbalance (explained in Section 5.1), it is important to overcome the limitation of conventional set-associative caches where each cache block can be placed at one of only a few places within a statically determined set. Thus, we propose to employ highly-associative caches, such as skewed-associative caches [53], ZCache [54], and V-Way cache [55], into hybrid cache design. These cache designs determine the physical location of cache blocks based on the hashed value of the block address and provide much higher associativity at low cost, both of which help to balance the load to each cache set. Therefore, highly-associative caches can improve the balance of write distribution across sets compared to conventional set-associative caches.

However, if we consider ZCache, a state-of-the-art highly-associative cache, simply applying it on top of hybrid caches introduces two important issues that can cause high energy overheads. This is because ZCache uses cuckoo hashing [73] to implement high associativity for replacement candidate selection. Similar to the skewed-associative caches, ZCache employs w hash functions to address w physical ways (i.e., w possible physical locations for each cache block); however, to effectively provide associativity higher than w with w physical ways, ZCache performs a *relocation process* during block insertion, where it (1) chooses one of the w locations for the block to be inserted, (2) displaces the victim cache block, (3) reinserts the displaced one into its other $w - 1$ possible locations, and (4) continues this process until we choose the block to be actually evicted from the cache (i.e., cuckoo hashing). This relocation process is particularly harmful to hybrid caches due to the following two reasons:

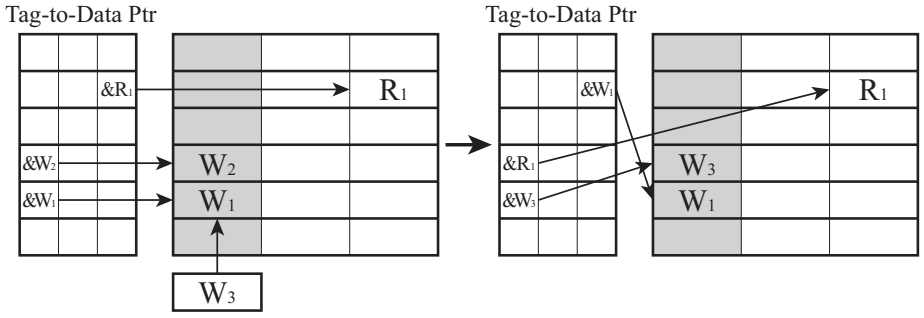
- **Relocation overhead**: The relocation process greatly increases the number of write operations to the cache. For example, under our ZCache configuration (see Section 5.3), every relocation incurs up to one extra read and write to the cache. This can be costly for hybrid caches where an STT-RAM write is an energy-consuming operation.

- **Interaction with hybrid caches**: Relocating cache blocks can disturb the data placement in hybrid caches. As we explained before, hybrid caches try to allocate write-intensive blocks into SRAM. However, the relocation process may move blocks in SRAM into STT-RAM and vice versa, which interferes with the data placement determined by the hybrid caches.

Figure 5.4a illustrates the two aforementioned problems. Let us assume that block W_3 is to be inserted at the slot where W_1 is stored and the cache selects W_2 as the victim for this insertion. ZCache performs cache replacement by replacing W_1 with W_3, R_1 with W_1, and W_2 with R_1 to insert W_3 into the cache while evicting W_2 (rather than



(a) Conventional ZCache



(b) ZCache with the tag-to-data pointer table

Figure 5.4 Intra-bank optimization. W_1 to W_3 are write-intensive blocks, while R_1 indicates a read-intensive block.

W_1). Not only does this incur a series of reads and writes to the cache for a single cache replacement, but also it makes W_1 (which is a write-intensive block) to be relocated from SRAM to STT-RAM after the replacement. Both of these can negatively affect the effectiveness of hybrid caches.

To address these problems, we propose to add a table for indirection from tags to data blocks, called *tag-to-data pointer table*. The tag-to-data pointer table contains the same number of entries as that of the tags (or data blocks) and stores one-to-one mapping information from tags to data blocks. With this table, each cache access is done by (1) searching for a matching tag, (2) *looking up the tag-to-data-pointer table to identify the location of the corresponding data block*, and (3) accessing the data block. Note that step 1 and 2 can be done in parallel, and thus, considering that the tag-to-data pointer table is much smaller than the tag array, it does not add any overhead to cache access latency.

Figure 5.4b shows the same example case with our tag-to-data pointer table. Instead of initiating a chain of replacements at the data array, our mechanism relocates only the tag-to-data pointers. Thus, it can directly replace W_2 with W_3 in the data array without additional data movement at the data array. Without the tag-to-data pointer table, this is not possible because there is only one SRAM block where W_3 can be inserted under the ZCache replacement policy, i.e., the slot where W_1 is located. Note that, from the replacement policy perspective, the resulting state of the ZCache with the tag-to-data pointer table is still exactly the same as the one without the tag-to-data pointer table.

The purpose of the tag-to-data pointer table is to decouple the physical location of data blocks in the data array from the location of tags. With this mechanism, data blocks can be stored anywhere in the data array without restriction, such as memory addresses, set indexing, or even the output of the hash function in ZCache. In other words, the tag-to-data pointer table *virtualizes* the physical location of data blocks in the data array.

Virtualizing the physical location of data blocks solves the two aforementioned inefficiencies. First, it allows the relocation process to be performed simply by relocating the entries in the tag-to-data pointer table without actually moving around the data blocks. This allows us to implement the relocation process without incurring any extra writes to the data array itself, which is beneficial for hybrid caches. Second, it ensures that the block placement done by hybrid caches is maintained even after the relocation process because it eliminates the need for actually moving the physical location of data blocks during the relocation process.

5.2.2 Inter-Bank Optimization

To reduce the inter-bank write imbalance (explained in Section 5.1), we propose inter-bank optimization in our architecture, which tries to evenly balance the number of writes per bank to globally improve the SRAM utilization. The high-level overview of our approach is to (1) profile the write intensity of each application by a new hardware module called *write intensity monitor*, (2) derive the write intensity of each bank based on application-level write intensity, and (3) periodically update the data placement by using our *write-intensity-aware data placement policy*, which aims at balancing the bank-level write intensity across all banks.

Write Intensity Monitor. As mentioned previously, we use application-level write intensity as a metric for evenly distributing the writes across different cache banks. This is because cache partitioning for distributed caches controls the placement of data for each application (e.g., share in Jigsaw) in distributed banks rather than enforcing the placement of each block. Fortunately, profiling write intensity of each application is straightforward since hybrid caches already track the information about write intensity of each cache block (e.g., cost model of PHC shown in Section 2.3.1) to guide block placement between SRAM and STT-RAM. Our idea is to utilize such information

to estimate the write intensity of each application. For example, we define the write intensity of an application as a sum of positive costs of cache blocks from the application:

$$AI_A = \frac{\sum \text{cost}_{bl}}{\text{size}_A}, \quad \forall bl \in BL_A, \text{ if } \text{cost}_{bl} > 0 \tag{5.1}$$

where AI_A denotes write intensity of application A, cost_{bl} denotes cost of block bl from application A, BL_A denotes the set of cache blocks loaded by application A, and size_A denotes the number of cache blocks allocated to application A by the cache partitioning scheme. Conceptually speaking, AI_A represents the average per-block cost of application A.

Based on the definition of AI, we now derive the write intensity of a bank. The key insight behind this is that for application A that stores part of its data to bank B, the amount of contribution of A to the write intensity of B is proportional to the proportion of A's data in B. Thus, we define the write intensity of bank B, or BI_B, as follows:

$$BI_B = \sum_A \left(AI_A \times \frac{\text{size}_B^A}{\text{bank\_size}} \right) \tag{5.2}$$

where size_B^A is the size of the A's partition in bank B and 'bank_size' is the capacity of a bank. Our data placement policy tries to balance this per-bank write intensity metric across all banks in the system.

Write-Intensity-Aware Data Placement. Our data placement scheme is based on existing schemes for data placement in distributed caches with cache partitioning (e.g., Jigsaw). First, our approach obtains the initial placement result from the existing scheme that does not consider write imbalance. Then, we refine this initial placement by iteratively swapping partitions from different banks in a way to improve the balance of per-bank write intensity. Since the write intensity of all banks should have the same value under perfect balance, the target per-bank write intensity is equal to *the average*

write intensity of all banks:

$$\overline{BI} = \frac{\sum_B \text{BI}_B}{N_B} \tag{5.3}$$

where N_B is the number of banks in the system.

To adjust the data placement so that all banks have write intensity close to $\overline{BI}$, we swap a portion of partitions from the bank that has too high write intensity to the one that has too low write intensity. First, we pick two (bank, application) pairs, (B_{high}, A_{high}) and (B_{low}, A_{low}), such that (1) the application in each pair stores part of its data in the associated bank, (2) the write intensity of B_{high} (or B_{low}) is higher (or lower) than $\overline{BI}$, and (3) the write intensity of A_{high} is higher than that of A_{low}. Then swapping some amount of A_{high}'s data in B_{high} with A_{low}'s data in B_{low} improves the balance of per-bank write intensity distribution because it exchanges write-intensive data in B_{high} (where write intensity is too high) with non-write-intensive data in B_{low} (where write intensity is too low).

The remaining question is on determining S, the amount of data to be swapped between B_{high} and B_{low}. If S is too small, it will have a minimal impact on the per-bank write intensity distribution. On the other hand, if S is too large, the write intensity of B_{high} (or B_{low}) may decrease (or increase) too much beyond $\overline{BI}$. In order to avoid such situations, we determine S based on the definition of per-bank write intensity. By definition, the swap operation changes the write intensity of B_{high} as follows:

$$\begin{aligned} \Delta BI_{B_{high}} = &- AI_{A_{high}} \times \frac{S}{\text{bank\_size}} \\ &+ AI_{A_{low}} \times \frac{S}{\text{bank\_size}} \end{aligned} \tag{5.4}$$

The first term on the right-hand side reflects that S amount of A_{high}'s data is moved out, while the second term adds the contribution of S amount of A_{low}'s data from B_{low}.

Similarly, we can calculate the difference in the write intensity of B_{low} as follows:

$$\begin{aligned}\Delta BI_{B_{low}} = &- AI_{A_{low}} \times \frac{S}{\text{bank\_size}} \\ &+ AI_{A_{high}} \times \frac{S}{\text{bank\_size}}\end{aligned} \tag{5.5}$$

Since our goal is to adjust the write intensity of both banks to be closer to $\overline{BI}$, we need to satisfy one of the following two equations:

$$BI_{B_{high}} + \Delta BI_{B_{high}} = \overline{BI} \tag{5.6}$$

$$BI_{B_{low}} + \Delta BI_{B_{low}} = \overline{BI} \tag{5.7}$$

Solving each equation for S results in the following two possible values for S:

$$S_{high} = \frac{\overline{BI} - BI_{B_{high}}}{-AI_{A_{high}} + AI_{A_{low}}} \times \text{bank\_size} \tag{5.8}$$

$$S_{low} = \frac{\overline{BI} - BI_{B_{low}}}{-AI_{A_{low}} + AI_{A_{high}}} \times \text{bank\_size} \tag{5.9}$$

Between the two, we choose the smaller one as S (i.e., $S = \min(S_{high}, S_{low})$) for more gradual refinement.[1] Conceptually speaking, this algorithm chooses the amount of exchange to be *proportional to the difference between the write intensity of B_{high} or B_{low} and $\overline{BI}$.*

Minimizing the Performance Overhead of Data Placement Adjustment. There are two sources of performance overhead in our data placement adjustment. First, moving cache blocks from one bank to another may cause a significant overhead even under a long reconfiguration period. Thus we do not perform proactive movement but adopt an on-demand migration technique in CDCS [74]. It adds an additional structure called shadow descriptor, which stores placement information from the previous reconfiguration period. The mechanism directs cache accesses to the old bank if the target cache

[1] If S is larger than the size of the partition allocated to A_{high} or A_{low} (i.e., $\text{size}_{B_{high}}^{A_{high}}$ and $\text{size}_{B_{low}}^{A_{low}}$), we use the smaller one between the two as the final S.

block is not yet moved to the new bank and triggers migration of the block *after the access*.

Second, if our placement algorithm moves cache blocks from a particular application farther from the core that runs the application, it may increase the on-chip network latency in accessing the cache blocks. Our approach mitigates such an overhead by rejecting swap operations that increase the hop-count between the core that run the application and cache blocks from that application by more than two hops (which we call *hop count constraint* λ). This prevents excessive increase in on-chip network latency.

Even with these two mechanisms, there are cases where the increase in on-chip network latency and traffic may offset the benefit of inter-bank optimization. This happens if workloads have low write intensity since the small amount of cache writes limits the energy reduction that can be achieved by distributing writes across different banks. Thus, Benzene monitors the sum of write intensity of all workloads in the system in every data placement period and *adaptively* disables inter-bank optimization if the write intensity is lower than a threshold ($\sum_A \mathrm{AI}_A < \rho$) [2]. In Section 5.4.5, we provide a sensitivity analysis of the hop count constraint and impact of the adaptive control of inter-bank optimization.

5.2.3 Other Optimizations

In our architecture, each bank is locally organized as a PHC, which was originally designed for centralized hybrid caches without cache partitioning. Thus, we propose two modifications to PHC to make it aware of cache partitioning.

Partitioning-Aware Set Sampling. To identify write-intensive data and update the PC-directed predictor, PHC has a hardware structure called *sampler*, which samples a

[2] We empirically determined the threshold (ρ) and set its value to 727 in our experiment.

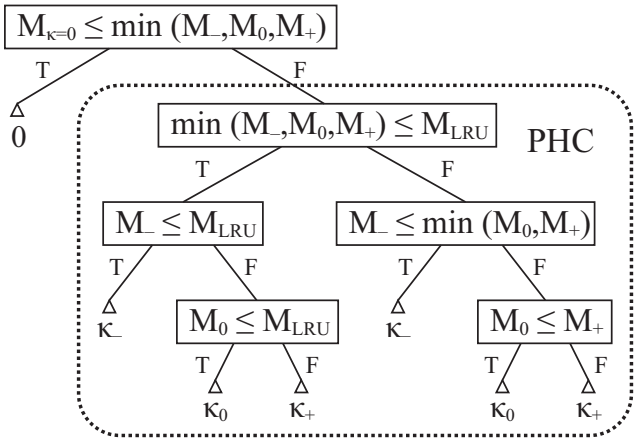


Figure 5.5 Decision tree for threshold adjustment in Benzene. Decision tree in PHC is shown in the dotted box

number of sets (i.e., set sampling [48,62,75]) and simulates cache replacement behavior of those sets to track the cost of sampled cache blocks. However, the sampler in PHC does not follow the cache partitioning decision that is enforced to the LLC in Benzene. This may cause mismatch between the cache replacement behavior emulated by the sampler and the actual cache replacement, which can degrade the prediction accuracy. Therefore, we organize a *partitioning-aware set sampler* where the sampler follows the same partitioning decision used in the LLC.

Better Threshold Adjustment. PHC periodically adjusts its write intensity threshold κ (see Section 2.3.1) to adapt to different levels of write intensity at runtime. Since the write intensity of most workloads slowly changes over time, PHC incrementally adjusts the threshold by incrementing/decrementing the threshold by a certain amount at the end of each period. For this purpose, PHC emulates the cache replacement behavior (particularly the number of misses) of the following four settings: the current threshold κ_0, its neighbor values κ_- ($< \kappa$) and κ_+ ($> \kappa$), and the plain LRU replacement without hybrid caches. Then it tries to use the lowest threshold that has fewer misses than the

LRU replacement. If all of them have more misses than the LRU replacement, it falls back to the threshold with the fewest misses. This is illustrated in the dotted box of Figure 5.5 (please refer to the original article [20] for more detailed information).

While this was sufficient in the original PHC, we observe that such gradual adjustment may cause PHC to fall into a local optimum under our architecture. This is because Benzene periodically alters its data placement to mitigate inter-bank write imbalance, which often changes the write intensity of each bank too rapidly to be followed by gradual threshold adjustment. Therefore, to escape from a potential local optimum, we add an extra sampler that constantly tracks the merit of using $\kappa = 0$ and use zero if it yields the lowest miss count among all candidate thresholds (i.e., the first branch of Figure 5.5).[3] This allows us to quickly escape from the current local optimum and restart the threshold exploration from the initial point.

5.3 Evaluation Methodology

We use zsim [76] to model a 64-tile manycore system with a distributed shared LLC (shown in Figure 5.3) and evaluate our architectural techniques based on it. Table 5.1 summarizes the detailed configuration of our baseline architecture. We use ZCache with 4 ways and 52 candidates. For distributed cache management, we use Jigsaw, which internally uses Vantage [52] as a partitioning scheme within each bank. We empirically determined the period of data placement in Jigsaw to 50 million cycles (i.e., $25\,\mathrm{ms}$) and the period of threshold update in PHC to 5 million cycles (i.e., $2.5\,\mathrm{ms}$).

We use CACTI [65] and NVSim [66] to model SRAM and STT-RAM, respectively, under 45nm technology with LOP devices. Table 5.2 shows the characteristics of an SRAM cache tile, an STT-RAM cache tile, and a hybrid cache tile used in our evaluation. Based on these parameters, we calculate ΔE_w and ΔE_r for PHC (see Section 2.3.1)

[3] Zero is a good starting point for threshold adjustment, because it is the breakeven point of allocating a block into SRAM instead of STT-RAM. In other words, allocating a block into SRAM reduces the energy consumption if and only if $\kappa > 0$.

Table 5.1 Configuration of the Simulated System

Component	**Configuration**
Core	64 Silvermont-like out-of-order cores, two-issue, 32-entry ROB, 2 GHz
L1 cache	Private, separate I/D, 32 KB each, 8-way set-associative, 64-B blocks, 3-cycle latency
L2 cache	Shared, 512 KB per tile (32 MB in total), 64-B blocks, PHC [20] with 1 SRAM way and 3 STT-RAM ways, distributed caches based on Jigsaw [22]
On-chip Network	8×8 mesh, 128-bit flits and links, X-Y routing, 3-cycle pipelined routers, 1-cycle links
Main memory	120-cycle zero-load latency, 4 channels, 12.8 GB/s per channel

Table 5.2 Characteristics of an LLC Tile

	SRAM	STT-RAM	Hybrid
Read Latency (cycles)	5	5	5/5*
Write Latency (cycles)	5	22	5/22*
Read Energy (nJ)	0.07	0.10	0.07/0.10*
Write Energy (nJ)	0.07	0.63	0.07/0.63*
Static Power (mW)	7.31	1.18	2.71

* Values for SRAM/STT-RAM, respectively.

Table 5.3 Applications from SPEC CPU2006

Write-Intensive Applications		Non-Write-Intensive Applications	
Application	WBPKI	Application	WBPKI
lbm	43.7	bzip2	3.3
soplex	22.0	hmmer	2.9
mcf	14.5	gromacs	2.4
zeusmp	11.6	povray	2.2
leslie3d	10.9	gobmk	2.2
GemsFDTD	10.8	calculix	2.0
astar	8.2	tonto	1.9
h264ref	6.3	namd	1.7
milc	6.1	cactusADM	1.5

to 20 and 1, respectively. Our latency and energy models include the overhead from additional hardware structures for Benzene (e.g., tag-to-data pointer table, extra sampler for $\kappa = 0$, etc.).

Our target is to reduce the energy consumption of hybrid caches for applications with write-intensive data (hybrid caches are already good at optimizing the energy consumption of non-write-intensive applications due to the low static power of STT-RAM). Thus, we choose 18 SPEC CPU2006 [77] applications (see Table 5.3) that have high L2 cache writebacks per kilo-instruction (WBPKI) and classify them into a write-intensive group (top nine applications with high WBPKI) and a non-write-intensive group (the other nine applications). Then, we synthesize twenty-five 64-application mixes [4] by varying the ratio of write-intensive applications in a mix from 100% (M1)

[4] In this work, we evaluate our architecture only with multi-programmed workloads. That said, Benzene can run multi-threaded workloads with no modifications to the partitioning scheme because the shared data placement used in Benzene is based on Jigsaw, which can handle both multi-programmed and multi-threaded workloads. Further improvements can be made by considering different types of shares (per-thread vs. per-process) in inter-bank optimization when running multi-threaded workloads, which is our future work.

Table 5.4 Workload characteristics

Workload	WBPKI	Workload	WBPKI	Workload	
M1	18.0	M11	8.7	M21	3.2
M2	21.1	M12	8.2	M22	3.2
M3	23.6	M13	8.4	M23	3.3
M4	19.5	M14	8.4	M24	3.2
M5	23.7	M15	8.5	M25	3.4
M6	12.4	M16	5.6		
M7	11.7	M17	5.4		
M8	11.6	M18	6.6		
M9	13.8	M19	5.3		
M10	12.8	M20	6.0		

to 0% (M25). Table 5.4 summarizes write-intensity of each workload. Each application mix is fast-forwarded for 20 billion instructions and then is simulated for 50 billion instructions in total. [5]

Our evaluation compares the following four configurations in terms of energy and performance. Unless otherwise specified, all results are normalized to the STT-RAM baseline. [6]

- **SRAM Baseline** represents a system with an SRAM-based L2 cache. This is essentially the same as Jigsaw.
- **STT-RAM Baseline** is the same as the SRAM baseline except that the L2 cache is constructed with STT-RAM only.
- **Benzene-Intra** is the proposed architecture with intra-bank optimization only.

[5] This may cause variation on executed instructions of each application, but the difference is less than 2.6%.

[6] The ideal baseline would be naïve combination of Jigsaw and PHC. However, it cannot be implemented because Jigsaw requires highly-associative caches while PHC is based on set-associative caches. The effectiveness of implementing hybrid caches based on highly-associative caches will be shown in Section 5.4.2.

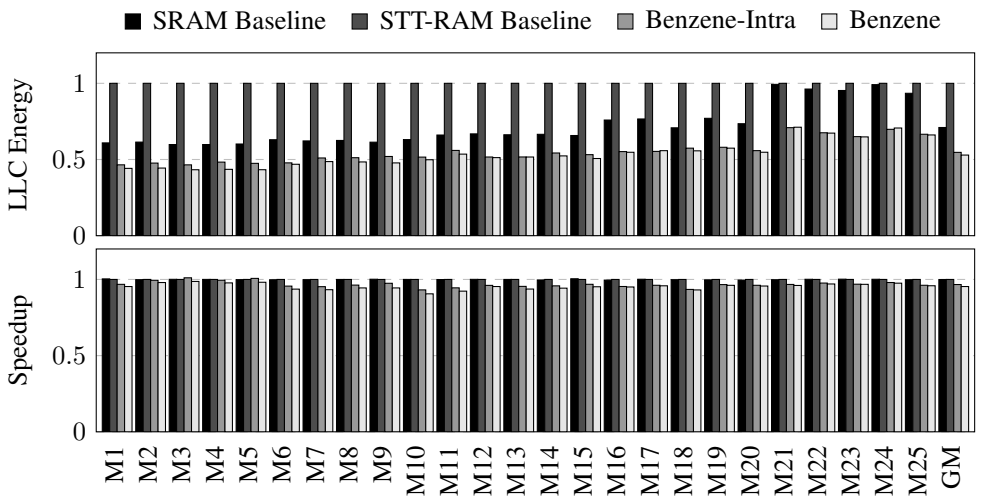


Figure 5.6 LLC energy consumption (above) and performance (below) of our architecture compared to the SRAM and STT-RAM baseline. All results are normalized to the STT-RAM baseline.

- **Benzene** includes both intra-bank and inter-bank optimizations. This is our final solution.

Benzene-Intra and Benzene include both of other optimization schemes (e.g., partitioning-aware set samping and better threshold adjustment).

5.4 Evaluation Results

5.4.1 Energy Consumption and Performance

Energy Consumption. Figure 5.6 shows the energy consumption of the LLC for four different configurations (normalized to the STT-RAM baseline). In addition, we provide more detailed analysis based on the LLC energy breakdown as shown in Figure 5.7 (averaged over all application mixes). Energy consumption of the predictor is very small (less than 0.1% in both Benzene-Intra and Benzene) and does not show up in the figure. From these results, we make the following observations.

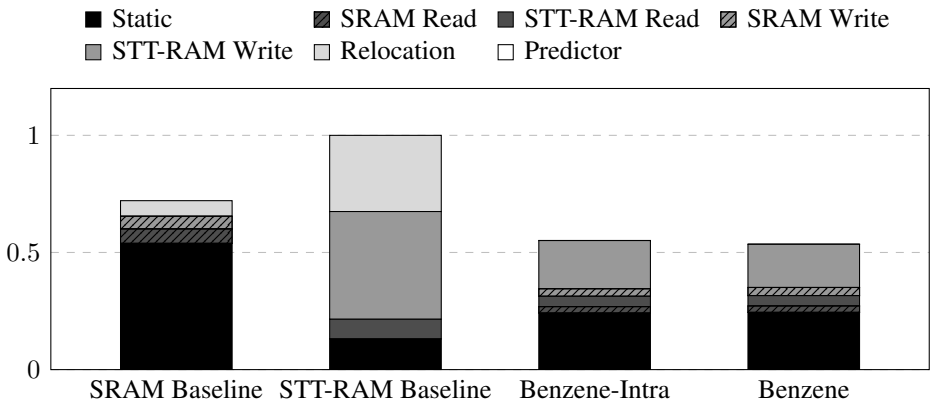


Figure 5.7 LLC energy breakdown. 'Relocation' represents the energy overhead of additional read/write operations caused by the cuckoo hashing mechanism of ZCache.

First, the SRAM baseline consumes 29.0% lower LLC energy with negligible performance differences. Although STT-RAM has lower static energy consumption (24.4% on average), this benefit is offset by higher write energy of STT-RAM. Due to this reason, the energy difference between the SRAM baseline and the STT-RAM baseline is larger in write-intensive workloads (towards M1). This implies the need for architectural techniques to reduce such write overheads especially in write-intensive workloads, which agrees with experimental results in many previous researches on STT-RAM caches [20, 43].

Second, Benzene-Intra reduces the LLC energy consumption by 45.3% compared to the STT-RAM baseline. This energy reduction comes from the following two reasons. First, hybrid caches effectively combine the benefit of lower static energy of STT-RAM and lower write energy of SRAM. Second, our tag-to-data pointer table eliminates the overhead of relocation (32.2% of LLC energy consumption in the STT-RAM baseline). Importantly, the latter enables us to construct highly-associative caches with STT-RAM in an energy-efficient manner, thereby realizing more even write distribution across different sets in a bank with very low overhead. We provide more analysis on the

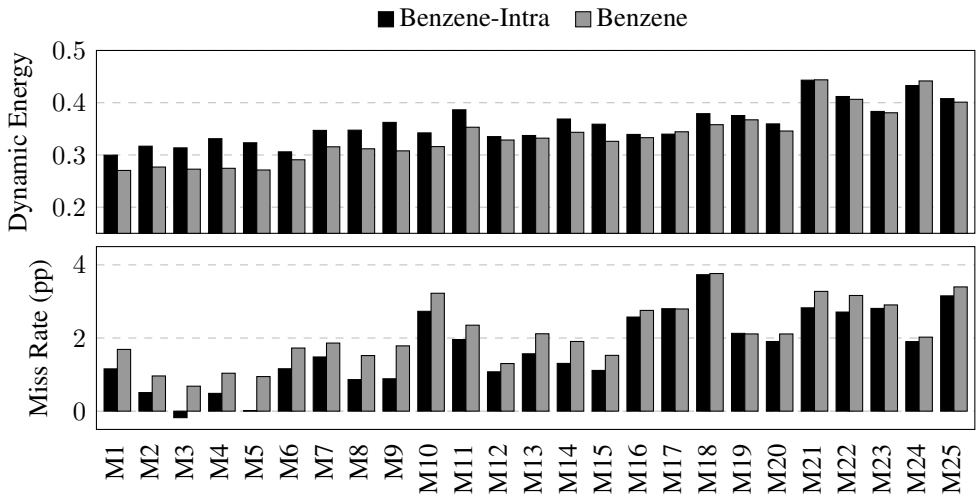


Figure 5.8 LLC dynamic energy (left) and miss rate (right) of Benzene-Intra and Benzene (normalized to STT-RAM baseline).

effectiveness of our intra-bank optimization in Section 5.4.2.

Third, our inter-bank optimization contributes an additional LLC energy reduction of 3.2% on average (up to 9.8%) on top of Benzene-Intra. More specifically, it reduces the write energy consumption by 10.4% on average. This is contributed by more even distribution of writes across different banks, which is important especially considering that only ¼ of the LLC is constructed with SRAM. We also observe that inter-bank optimization is more effective when the workloads are write-intensive (e.g., M1 to M5 reduces LLC energy consumption by 7.5% on average) because balancing writes across different banks is more important if workloads generate a large number of writes. Section 5.4.3 analyzes the impact of our inter-bank optimization in detail.

Figure 5.8 shows the dynamic energy consumption of Benzene-Intra and Benzene, normalized to the STT-RAM baseline. This clearly shows that the reduction in dynamic energy consumption is the key source of LLC energy reduction.

Performance. Figure 5.6 also compares the system performance of the baseline and our architecture. We use the sum of IPC of all applications (weighted speedup values have a similar trend) as the metric of performance comparison. We observe that our intra-bank optimization has 3.3% performance drop compared to the STT-RAM baseline. This is caused by inaccurate threshold adjustment of PHC for distributed caches (note that PHC is originally designed for centralized caches), which leads to 1.7 percentage points (pp) increase in miss rate on average as shown in Figure 5.8. Our inter-bank optimization adds a small performance overhead to it (1.2% on average), which comes from the increased network latency caused by our write-intensity-aware data placement. Note that there is a trade-off relationship between energy reduction and performance overhead and our hop count constraint and adaptive inter-bank optimization allows this performance overhead to be tuned according to system requirement.

In summary, Benzene saves LLC energy by 47.1% with only 4.5% performance overhead.[7] Although the benefit from inter-bank optimization may be offset by performance drop, inter-bank optimization is still useful in terms of energy reduction, especially when the workloads are write-intensive.

5.4.2 Analysis of Intra-bank Optimization

Figure 5.9 shows the distribution of writes across 2,048 sets in a set-associative cache (same as the one in Figure 5.1) and a highly-associative cache. The result is obtained from a 64-process workload running on a 64-core system as in ZCache [54]. In this figure, a histogram with more concentrated bars indicates more uniform distribution of writes across different cache sets. As shown in the figure, using a highly-associative

[7] According to the modeling result from McPAT [78] for the baseline architecture (Silvermont-like core), LLC consumes 15% of total processor energy and the static energy of the rest of the components (excluding LLC) takes 13% of the total processor energy. If we combine these numbers with the energy reduction (26% over SRAM Baseline) and performance overhead (4.5%) of Benzene, Benzene is still expected to be beneficial in terms of total energy consumption (15% * 0.26 - 13% * 0.045 = 3.3% reduction in terms of total processor energy consumption).

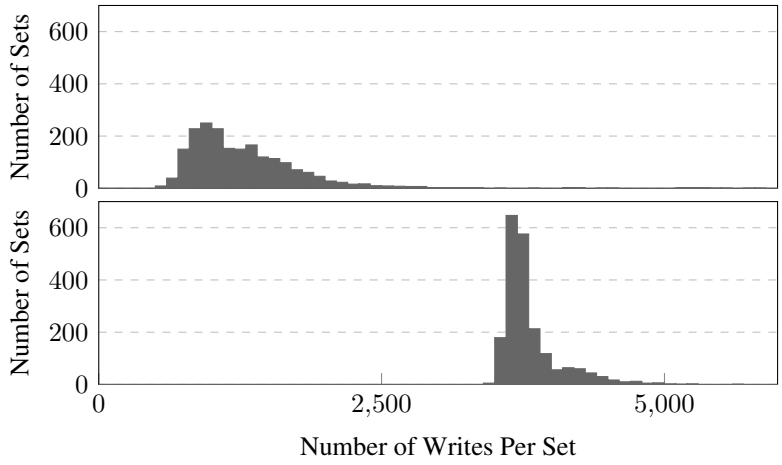


Figure 5.9 Histogram of SRAM write distribution across different sets in a set-associative cache (above) and a highly-associative cache with a tag-to-data pointer table (below).

cache greatly improves the set-level write distribution. Quantitatively speaking, the standard deviation of SRAM writes per set is 2.86 times lower in a highly-associative cache than in a set-associative cache for this particular example. In the end, this balanced write distribution across sets improves SRAM utilization, e.g., the ratio of SRAM write increases from 25.7% to 72.8%.

Note that it is the tag-to-data pointer table that enables such optimization *in an energy-efficient manner*. Simply applying a highly-associative cache on top of a hybrid cache does not improve the LLC energy efficiency because of the following two reasons. First, without the tag-to-data pointer table, the relocation overhead of ZCache incurs 32.5% of LLC energy as explained in the previous subsection. Second, 20.7% and 37.9% of total relocation operations move write-intensive data from SRAM to STT-RAM and non-write-intensive data from STT-RAM to SRAM, respectively (as exemplified in Figure 5.4), which should not be moved. This makes hybrid cache management inefficient.

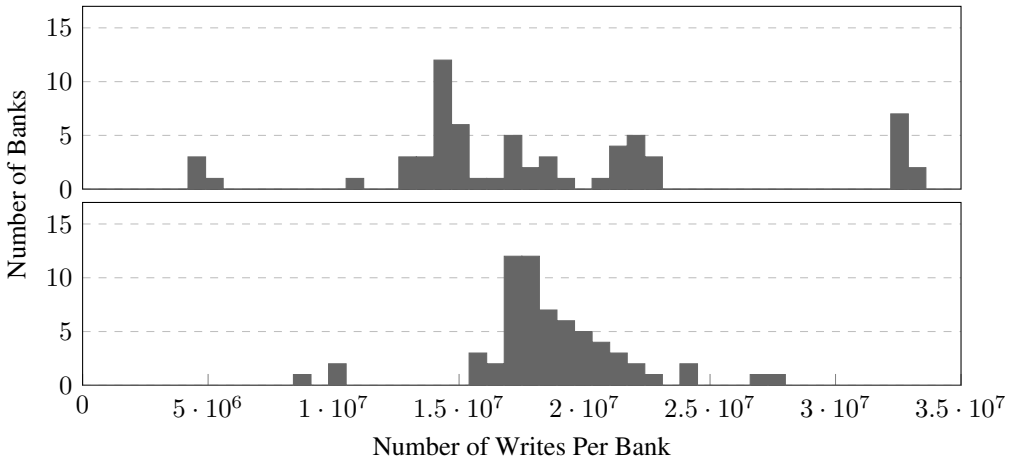


Figure 5.10 Histogram of write distribution across different banks in a distributed cache before (above) and after (below) applying our inter-bank optimization.

5.4.3 Analysis of Inter-bank Optimization

Figure 5.10 shows the effectiveness of our inter-bank optimization on balancing the distribution of writes across 64 banks in a distributed cache architecture. The figure shows the distribution obtained from application mix M4. Similarly to the previous subsection, a histogram with more concentrated bars is more desirable in terms of write balance.

As shown in the figure, writes in the conventional distributed cache are concentrated to a few banks. In particular, we observe that some banks receive more than 7.6x writes than the average writes per bank, which can degrade the effectiveness of hybrid caches. On the contrary, our inter-bank optimization achieves more even distribution by adjusting the placement of data in a way to balance writes per bank across the distributed cache. As a result, our approach reduces the standard deviation of writes per bank by 46.6% on average compared to the conventional distributed cache.

Better balance in write distribution directly affects SRAM utilization in distributed

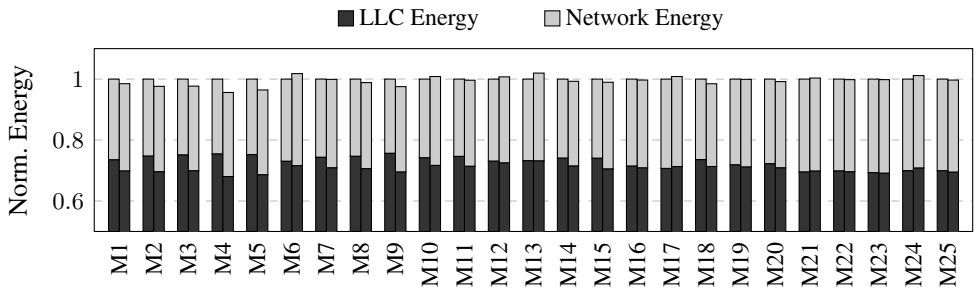


Figure 5.11 Normalized energy consumption of LLC and on-chip network in Benzene-Intra (left) and Benzene (right).

hybrid caches. We observe that Benzene improves the ratio of writes to SRAM in a distributed hybrid cache by 4.7pp on average (up to 13.1pp) compared to Benzene-Intra. Since both Benzene and Benzene-Intra have almost the same number of writes in total, this indicates that Benzene puts more writes to SRAM and thus has higher SRAM utilization. This directly contributes to 6.5% (up to 17.1%) of the dynamic energy reduction shown in Figure 5.8.

5.4.4 Impact of Inter-Bank Optimization on Network Energy

As mentioned in Section 5.2.2, inter-bank optimization increases data accesses over on-chip network. Figure 5.11 compares Benzene-Intra and Benzene in terms of the total energy consumption including both LLC and on-chip network (the network energy is modeled by DSENT [79]). We do not compare other configurations because intra-bank optimization does not affect bank placement decision made by Jigsaw (i.e., no difference in network traffic between the baselines and Benzene-Intra). We observe that, even though inter-bank optimization increases the network energy consumption by 9.1% on average, the total energy is still reduced by 2.8% (up to 4.4%). For write-intensive workloads, the benefit from write energy reduction due to inter-bank optimization is larger than the additional overhead of network energy (note that the increase in network energy is controlled by hop count constraint). For non-write-intensive workloads,

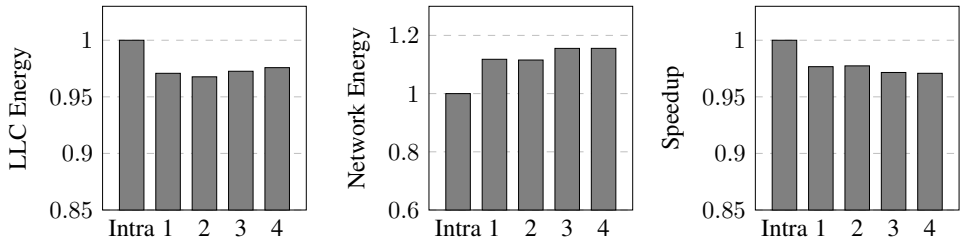


Figure 5.12 Impact of the hop count constraint on LLC energy (left), network energy (middle), and performance (right). The results are normalized to Benzene-Intra (Intra in the figure).

Benzene simply disables inter-bank optimization (i.e., adaptive inter-bank optimization), which makes it as efficient as Benzene-Intra. Therefore, we conclude that the inter-bank optimization reduces energy consumption, particularly for write-intensive workloads, even if we take the increased network energy consumption into account.

5.4.5 Sensitivity Analysis

Hop Count Constraint. Figure 5.12 shows the LLC/network energy consumption and the performance of Benzene under different hop count constraint λ (see Section 5.2.2) for our inter-bank optimization. In this figure, we always enable inter-bank optimization (i.e., no adaptive inter-bank optimization) to show the sole impact of hop count constraint. As shown in the figure, the looser the constraint is, the higher the performance drop and network energy consumption are. On the other hand, LLC energy consumption is reduced the most when $\lambda = 2$ because of the following reasons. When $\lambda = 1$, the hop count constraint is too restrictive to evenly balance the write distribution across all banks; when $\lambda \geq 3$, it degrades performance too much and thus increases the static energy consumption, which offsets the benefit of dynamic energy reduction. $\lambda = 2$ balances between the two and achieves maximal energy reduction, which is why we used $\lambda = 2$ throughout the experiments.

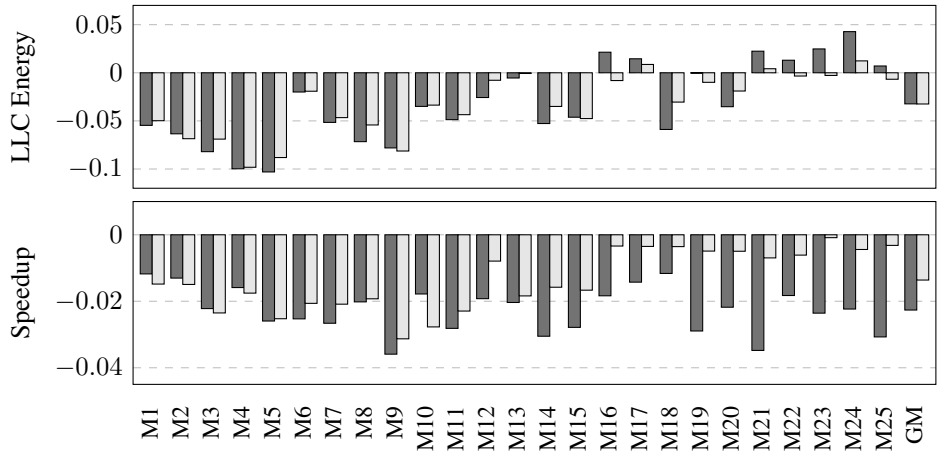


Figure 5.13 The impact of adaptive inter-bank optimization on LLC energy consumption and performance. Benzene without adaptive inter-bank optimization (left) and Benzene (right), compared to Benzene-Intra.

Adaptive Inter-Bank Optimization. Figure 5.13 shows the impact of inter-bank optimization on LLC energy consumption and performance. As shown in the figure, adaptive inter-bank optimization disables inter-bank write distribution for non-write-intensive workloads, where inter-bank optimization is not worthwhile in terms of energy reduction. In other words, adaptive inter-bank optimization allows us to combine the best of Benzene-Intra and Benzene according to the workload characteristics.

Other Optimizations. Figure 5.14 shows the impact of other optimization techniques, i.e., partitioning-aware sampling (**P** in the figure) and better threshold adjustment (**T** in the figure). When both **P** and **T** are applied (i.e., **P+T**), they improve the LLC miss rate by 0.3pp and the performance by 0.6% compared to the Benzene-Intra without them.

5.4.6 Implementation Overhead

Benzene introduces very small overhead to existing distributed hybrid cache architectures with distributed cache partitioning. The intra-bank optimization adds a tag-to-data

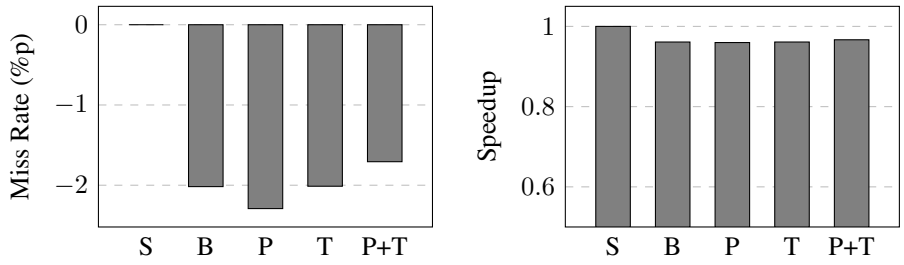


Figure 5.14 The impact of partitioning-aware sampling and better threshold adjustment on LLC miss rate (left) and speedup (right). The figure shows comparison among the STT-RAM baseline (**S**), Benzene-Intra without partitioning-aware sampling or threshold adjustment (**B**), Benzene-Intra with partitioning-aware sampling (**P**), Benzene-Intra with threshold adjustment (**T**), and Benzene-Intra with partitioning-aware sampling and threshold adjustment (**P+T**).

pointer table, which has 8192 13-bit entries per LLC slice to establish one-to-one mapping from tags to data blocks. The inter-bank optimization mostly utilizes existing hardware (e.g., the write intensity predictor from PHC to estimate per-bank write intensity) and thus has a negligible area overhead. Partitioning-aware sampling adds a partition field to each sampler entry and a vantage controller to each sampler, which introduces $1\,\mathrm{KB}$ storage overhead per sampler. Better threshold adjustment introduces an additional sampler for $\kappa = 0$, which adds $0.66\,\mathrm{KB}$ per LLC slice. In total, Benzene introduces $17\,\mathrm{KB}$ of storage overhead per LLC tile, or 2.1 bits per block.

Benzene runs intra-bank optimization periodically every 50 million cycles ($25\,\mathrm{ms}$ in our implementation). As in Jigsaw, it is performed by software on one of the cores in the system and the extra performance overhead caused by this is only 0.12%.

5.5 Summary

In this work, we proposed utilizing SRAM/STT-RAM hybrid caches in distributed cache architecture for manycore systems. Since previous hybrid cache techniques do not consider distributed cache, they are not scalable and underutilize the scarce SRAM

resource which determines efficiency of hybrid caches. Our proposed architecture, Benzene, improves SRAM utilization and maximizes efficiency of hybrid cache on top of distributed cache architecture by distributing write-intensive data evenly in two different levels (intra-bank and inter-bank).

Chapter 6

Conculsion

This dissertation proposed design techniques to build on-chip cache energy efficient using STT-RAM. Since STT-RAM suffers from poor write characteristics, the proposed techniques try to reduce the number of STT-RAM writes or make write characteristics of STT-RAM better.

First, we proposed an energy-efficient exclusive LLC architecture based on STT-RAM to take advantage of capacity benefit of exclusive caches and low static power of STT-RAM. The key challenge in designing such an architecture is the increased amount of LLC writes, which is detrimental for STT-RAM caches that show poor write characteristics. To address this issue, our architecture is composed of (1) a bypassing scheme to avoid write overhead for far reuse cache blocks, (2) a hybrid cache architecture that reduces write energy consumption for near reuse blocks with a small SRAM cache, and (3) a reuse distance predictor that realizes such decision in a cost-effective manner. Our evaluations show that the proposed architecture reduces LLC energy by 55% with slight improvement in energy efficiency of main memory and system performance.

Second, we analyze and evaluate cache architectures using volatile STT-RAM. Volatile STT-RAM, which can be designed by reducing the thermal stability of the cells, has been proposed to address poor write characteristics which is a main drawback of conventional non-volatile STT-RAM. However, to utilize volatile STT-RAM, ECC with periodic scrubbing is mandatory due to its high retention failure rate. We evaluate volatile STT-RAM LLC with various ECC strength, scrubbing period, and target failure rate configurations. Based on the evaluation, we reveal that ECC and scrubbing overhead hides the benefits of volatile STT-RAM over non-volatile STT-RAM in most configurations. For a short scrubbing period, the scrubbing overhead offsets the benefit from using volatile STT-RAM. Using a long scrubbing period makes the scrubbing overhead negligible but it increases STT-RAM write energy. In terms of ECC, only a weak ECC (SECDED) is beneficial since a stronger ECC suffers from its overhead. Moreover, varying the target failure rate affects LLC energy very slightly. In conclusion, weak ECC (SECDED) combined with a moderate scrubbing period (100ms) improves energy efficiency at only a negligible performance degradation.

Third, we proposed Benzene, a scalable STT-RAM cache architecture for manycore systems. Our key observation is that (1) distributed cache architectures exhibit significant write imbalance in two different levels (intra-bank and inter-bank) and (2) such write imbalance leads to underutilization of SRAM resources in hybrid caches, thereby leaving room for energy efficiency improvement. Benzene leverages this observation and proposes two architectural optimizations. First, intra-bank optimization reduces write imbalance at the cache set level by exploiting highly-associative cache design with a new hardware structure called tag-to-data pointer table. Second, inter-bank optimization achieves better write distribution across all banks in the distributed cache through our write-intensity-aware data placement policy. Our evaluation results show that, with this two optimizations, Benzene reduces the variance in the number of writes within and across the banks by 51.3% and 46.6%, respectively, thereby achieving a

47.1% reduction in LLC energy consumption. We believe that Benzene can facilitate *scalable* application of STT-RAM caches in manycore systems.

Bibliography

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang, "SkyLake-SP: A 14nm 28-core Xeon® processor," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2018, pp. 34–36.

[3] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement," in *Proceedings of the Design Automation Conference*, 2008, pp. 554–559.

[4] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L^3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2013, pp. 143–154.

[5] Y. Zheng, B. T. Davis, and M. Jordan, "Performance evaluation of exclusive cache hierarchies," in *Proceedings of the International Symposium on Performance*

Analysis of Systems and Software, 2004, pp. 89–96.

[6] *Family 16h Models 00h - 0Fh AMD Opteron™ Processor Product Data Sheet*, Advanced Micro Devices, Jun. 2013.

[7] *Software Optimization Guide for AMD Family 17h Processors*, Advanced Micro Devices, Jun. 2017.

[8] N. Kim, J. Ahn, W. Seo, and K. Choi, "Energy-efficient exclusive last-level hybrid caches consisting of SRAM and STT-RAM," in *Proceedings of the International Conference on Very Large Scale Integration*, 2015, pp. 183–188.

[9] C. W. Smullen IV, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011, pp. 50–61.

[10] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STTRAM scaling and retention failure," *Intel® Technology Journal*, vol. 17, no. 1, pp. 54–75, May 2013.

[11] C. W. Smullen IV, "Designing giga-scale memory systems with STT-RAM," Ph.D. dissertation, University of Virginia, 2011.

[12] N. Kim and K. Choi, "Exploration of trade-offs in the design of volatile STT-RAM cache," *Journal of Systems Architecture*, vol. 71, pp. 23 – 31, Nov. 2016.

[13] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs," in *Proceedings of the Design Automation Conference*, 2012, pp. 243–252.

[14] Z. Sun, X. Bi, H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu, "Multi retention level STT-RAM cache designs with a dynamic refresh scheme," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 329–338.

[15] N. Strikos, V. Kontorinis, X. Dong, H. Homayoun, and D. Tullsen, "Low-current probabilistic writes for power-efficient STT-RAM caches," in *Proceedings of the International Conference on Computer Design*, 2013, pp. 511–514.

[16] K. Swaminathan, R. Pisolkar, C. Xu, and V. Narayanan, "When to forget: A system-level perspective on STT-RAMs," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2012, pp. 311–316.

[17] B. Del Bel, J. Kim, C. Kim, and S. Sapatnekar, "Improving STT-MRAM density through multibit error correction," in *Proceedings of the Design, Automation and Test in Europe*, 2014, pp. 1–6.

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64™-processor: A 64-core SoC with mesh interconnect," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2008, pp. 88–598.

[19] G. Chrysos, "Intel® Xeon Phi coprocessor (codename Knights Corner)," in *Hot Chips Symposium*, 2012, pp. 1–31.

[20] J. Ahn, S. Yoo, and K. Choi, "Prediction hybrid cache: An energy-efficient STT-RAM cache architecture," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 940–951, Mar. 2016.

[21] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 34–45.

[22] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 213–224.

[23] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011, pp. 219–230.

[24] N. Kim, J. Ahn, K. Choi, D. Sanchez, D. Yoo, and S. Ryu, "Benzene: An energy-efficient distributed hybrid cache architecture for manycore systems," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 1, pp. 10:1–10:23, Mar. 2018.

[25] A. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. Butler, P. Visscher *et al.*, "Basic principles of STT-MRAM cell operation in memory arrays," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074001, Jan. 2013.

[26] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM," in *International Electron Devices Meeting Technical Digest*, 2005, pp. 459–462.

[27] H. Sun, C. Liu, N. Zheng, T. Min, and T. Zhang, "Design techniques to improve the device write margin for MRAM-based cache memory," in *Proceedings of the Great Lakes Symposium on VLSI*, 2011, pp. 97–102.

[28] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Transactions on Reliability*, vol. 39, no. 1, pp. 114–122, Apr. 1990.

[29] *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 11h Processors*, Advanced Micro Devices, Jul. 2008.

[30] Q. Li, Y. He, J. Li, L. Shi, Y. Chen, and C. J. Xue, "Compiler-assisted refresh minimization for volatile STT-RAM cache," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2169–2181, Aug 2015.

[31] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proceedings of the International Conference on Computer-Aided Design*, 2009, pp. 264–268.

[32] X. Bi, Z. Sun, H. Li, and W. Wu, "Probabilistic design methodology to improve run-time stability and performance of STT-RAM caches," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 88–94.

[33] T. Zheng, J. Park, M. Orshansky, and M. Erez, "Variable-energy write STT-RAM architecture with bit-wise write-completion monitoring," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2013, pp. 229–234.

[34] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proceedings of the Design, Automation and Test in Europe*, 2009, pp. 737–742.

[35] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009, pp. 239–249.

[36] Q. Li, J. Li, L. Shi, C. J. Xue, and Y. He, "MAC: Migration-aware compilation for STT-RAM based hybrid cache in embedded systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2012, pp. 329–342.

[37] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2012, pp. 237–242.

[38] J. Li, L. Shi, C. J. Xue, C. Yang, and Y. Xu, "Exploiting set-level write non-uniformity for energy-efficient NVM-based hybrid cache," in *Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia*, 2011, pp. 19–28.

[39] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache," in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2012, pp. 109–118.

[40] J. Li, C. J. Xue, and Y. Xu, "STT-RAM based energy-efficiency hybrid cache for CMPs," in *Proceedings of the International Conference on Very Large Scale Integration*, 2011, pp. 31–36.

[41] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014, pp. 13–24.

[42] J. Wang, X. Dong, and Y. Xie, "OAP: An obstruction-aware cache management policy for STT-RAM last-level caches," in *Proceedings of the Design, Automation and Test in Europe*, 2013, pp. 847–852.

[43] J. Ahn, S. Yoo, and K. Choi, "DASCA: Dead write prediction assisted STT-RAM cache architecture," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014, pp. 25–36.

[44] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Apr. 2018.

[45] J. Sim, J. Lee, M. K. Qureshi, and H. Kim, "FLEXclusion: Balancing cache capacity and on-chip bandwidth via flexible exclusion," in *Proceedings of the International Symposium on Computer Architecture*, 2012, pp. 321–332.

[46] A. Jaleel, J. Nuzman, A. Moga, S. C. S. Jr, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015, pp. 343–353.

[47] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings of the International Symposium on Computer Architecture*, 2001, pp. 144–154.

[48] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the International Symposium on Microarchitecture*, 2010, pp. 175–186.

[49] D. Chiou, L. Rudolph, S. Devadas, and B. S. Ang, "Dynamic cache partitioning via columnization," Computation Structures Group Memo 430, MIT, 1999.

[50] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the International Symposium on Microarchitecture*, 2006, pp. 423–432.

[51] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 174–183.

[52] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proceedings of the International Symposium on Computer Architecture*, 2011, pp. 57–68.

[53] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 169–178.

[54] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *Proceedings of the International Symposium on Microarchitecture*, 2010, pp. 187–198.

[55] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: Demand-based associativity via global replacement," in *Proceedings of the International Symposium on Computer Architecture*, 2005, pp. 544–555.

[56] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "The reuse cache: Downsizing the shared last-level cache," in *Proceedings of the International Symposium on Microarchitecture*, 2013, pp. 310–321.

[57] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *Proceedings of the International Symposium on Microarchitecture*, 2003, pp. 55–66.

[58] ——, "Optimizing replication, communication, and capacity allocation in cmps," in *Proceedings of the International Symposium on Computer Architecture*, 2005, pp. 357–368.

[59] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 430–441.

[60] P. Petoumenos, G. Keramidas, and S. Kaxiras, "Instruction-based reuse-distance prediction for effective cache management," in *Proceedings of the International Conference on Systems, Architectures, Modeling and Simulation*, 2009, pp. 49–58.

[61] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2003, pp. 245–257.

[62] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proceedings of the International Symposium on Computer Architecture*, 2006, pp. 167–178.

[63] *MacSim Simulator*. [Online]. Available: https://github.com/gthparch/macsim

[64] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proceedings of the International Symposium on Computer Architecture*, 2011, pp. 81–92.

[65] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[66] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[67] Y. Chen, X. Wang, H. Li, H. Xi, Y. Yan, and W. Zhu, "Design margin exploration of spin-transfer torque RAM (STT-RAM) in scaled technologies," *IEEE Transactions*

on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 12, pp. 1724–1734, Dec. 2010.

[68] Y. Zhang, W. Wen, and Y. Chen, "The prospect of STT-RAM scaling from readability perspective," *IEEE Transactions on Magnetics*, vol. 48, no. 11, pp. 3035–3038, Nov. 2012.

[69] A. Phansalkar, A. Joshi, and L. K. John, "Subsetting the SPEC CPU2006 benchmark suite," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 69–76, Mar. 2007.

[70] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation," in *Proceedings of the International Symposium on Microarchitecture*, 2004, pp. 81–92.

[71] "Calculating memory system power for DDR3," Micron Technology, Tech. Rep. TN-41-01, 2007.

[72] T. Kishi, H. Yoda, T. Kai, T. Nagase, E. Kitagawa, M. Yoshikawa, K. Nishiyama, T. Daibou, M. Nagamine, M. Amano *et al.*, "Lower-current and fast switching of a perpendicular TMR for high speed and high density spin-transfer-torque MRAM," in *International Electron Devices Meeting Technical Digest*, 2008, pp. 1–4.

[73] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[74] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *Proceedings of the International Symposium in High Performance Computer Architecture*, 2015, pp. 538–550.

[75] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *Proceedings of the International Conference on Computer Design*, 2007, pp. 245–250.

[76] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 475–786.

[77] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[78] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework formulticore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 1, pp. 5:1–5:29, Apr. 2013.

[79] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proceedings of the International Symposium on Networks on Chip*, 2012, pp. 201–210.

초록

지난 수십 년간 '메모리 벽' 문제를 해결하기 위해 온 칩 캐시의 크기는 꾸준히 증가해왔다. 하지만 지금까지 캐시에 주로 사용되어 온 메모리 기술인 SRAM은 낮은 집적도와 높은 대기 전력 소모로 인해 큰 캐시를 구성하는 데에는 적합하지 않다. 이러한 SRAM의 단점을 보완하기 위해 더 높은 집적도와 낮은 대기 전력을 소모하는 새로운 메모리 기술인 STT-RAM으로 SRAM을 대체하는 것이 제안되었다. 하지만 STT-RAM은 데이터를 쓸 때 많은 에너지와 시간을 소비하기 때문에 단순히 SRAM을 STT-RAM으로 대체하는 것은 오히려 캐시 에너지 소비를 증가시킨다. 이러한 문제를 해결하기 위해 본 논문에서는 STT-RAM을 이용한 에너지 효율적인 캐시 설계 기술들을 제안한다.

첫 번째, 배타적 캐시 계층 구조에서 STT-RAM을 활용하는 방법을 제안하였다. 배타적 캐시 계층 구조는 계층 간에 중복된 데이터가 없기 때문에 포함적 캐시 계층 구조와 비교하여 더 큰 유효 용량을 갖지만, 배타적 캐시 계층 구조에서는 상위 레벨 캐시에서 내보내진 모든 데이터를 하위 레벨 캐시에 써야 하므로 더 많은 양의 데이터를 쓰게 된다. 이러한 배타적 캐시 계층 구조의 특성은 쓰기 특성이 단점인 STT-RAM을 함께 활용하는 것을 어렵게 한다. 이를 해결하기 위해 본 논문에서는 재사용 거리 예측을 기반으로 하는 SRAM/STT-RAM 하이브리드 캐시 구조를 설계하였다.

두 번째, 비휘발성 STT-RAM을 이용해 캐시를 설계할 때 고려해야 할 점들에 대해 분석하였다. STT-RAM의 비효율적인 쓰기 동작을 줄이기 위해 다양한 해결법들이 제안되었다. 그중 한 가지는 STT-RAM 소자가 데이터를 유지하는 시간을 줄여 (휘발성 STT-RAM) 쓰기 특성을 향상하는 방법이다. STT-RAM에 저장된 데이터를 잃는 것은 확률적으로 발생하기 때문에 저장된 데이터를 안정적으로 유지하기 위해서는 오류 정정 부호(ECC)를 이용해 주기적으로 오류를 정정해주어야 한다.

본 논문에서는 STT-RAM 모델을 이용하여 휘발성 STT-RAM 설계 요소들에 대해 분석하였고 실험을 통해 해당 설계 요소들이 캐시 에너지와 성능에 주는 영향을 보여주었다.

마지막으로, 매니코어 시스템에서의 분산 하이브리드 캐시 구조를 설계하였다. 단순히 기존의 하이브리드 캐시와 분산캐시를 결합하면 하이브리드 캐시의 효율성에 큰 영향을 주는 SRAM 활용도가 낮아진다. 따라서 기존의 하이브리드 캐시 구조에서의 에너지 감소를 기대할 수 없다. 본 논문에서는 분산 하이브리드 캐시 구조에서 SRAM 활용도를 높일 수 있는 두 가지 최적화 기술인 뱅크-내부 최적화와 뱅크간 최적화 기술을 제안하였다. 뱅크-내부 최적화는 highly-associative 캐시를 활용하여 뱅크 내부에서 쓰기 동작이 많은 데이터를 분산시키는 것이고 뱅크간 최적화는 서로 다른 캐시 뱅크에 쓰기 동작이 많은 데이터를 고르게 분산시키는 최적화 방법이다.

주요어: 컴퓨터 구조, 메모리 시스템, 캐시, 에너지 효율, 비휘발성 메모리, STT-RAM
학번: 2013-20751